University of Alberta

Analysis of Software Measures by Self Organizing Maps

by

Xiao Bai

A thesis submitted to the faculty of Graduate Studies and Research in partial fulfillment

of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta

Spring, 2001

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-60409-8

Canada

University of Alberta

Library Release Form

Name of Author:	Xiao Bai
Title of Thesis:	Analysis of Software Measures by Self Organizing Maps
Degree:	Master of Science
Year this Degree Granted:	2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Xiao Bai

Xiao Bai
8911, 112 St., Apt. 01A
Edmonton, Alberta
Canada, T6G 2C5

Date: Mar 27, 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Analysis of Software Measures by Self Organizing Maps** submitted by **Xiao Bai** in partial fulfillment of the requirements for the degree of Master of Science.

Witold Pedrycz

Marek Reformat

Eleni Stroulia

Date: March 27, 2001

Abstract

Software measurements provide a quantitative method in software engineering by which the software development process can be evaluated and controlled. Software measures are complex, involving highly dimensional data and are difficult to analyze by traditional statistical means. Self Organizing Maps(SOM), a particular neural-computing approach, on the other hand can alternatively be applied to analyze software measures. SOM can be used to visualize highly-dimensional data by means of a collection of maps so that one can readily determine the clustering and feature relationships in the software measure data. This gives one a direct and easy means by which to capture the characteristics of the data. Thus the software managers and developers, by learning how the software modules are related and clustered may get a thorough and much better insight into their software projects.

Acknowledgement

This thesis has been carried out as a student in the Department of Electrical and Computer Engineering at the University of Alberta during the time from the year 1999 to the date of submission of this thesis. I wish to thank most of all my supervisor, Professor Witold Pedrycz, for guidance consisting of numerous recommendations and ideas that have had a major effect on the contents of this thesis.

In addition I also wish to thank the other professors in our Quantitative Software Engineering Lab such as Dr. Giancarlo Succi, Dr. Petr Musilek and Dr. M. Reformat, all of whom have given me valuable suggestions during my research.

I must also thank the visiting professors and other graduate students in our lab. Namely, Dr. Brouwer for reviewing my thesis, Dr. Chun for discussing new ideas with me, students Marlene and Lino for testing my SOM Tool software, and students Danny, Milorad, Raymond and Alexander for providing me with experimental data.

Finally, the financial support from the ASERC(Alberta Software Engineering Research Consortium) is also gratefully acknowledged.

Table of Contents

1 Introduction 1

1.1 Software Measurements 1

1.2 Software Measure Analysis 1

1.3 Self Organizing Maps 2

1.4 Research Objectives 2

1.5 Thesis Organization 3

2 Software Measurements 4

2.1 General Concept of Measurements 4

2.2 Motivation of Software Measurements 4

2.2.1 Fast Growing Complexity of Software 4

2.2.2 Software Engineering 4

2.2.3 Application of Software Measurements 4

2.3 Classification of Software Measures 5

2.3.1 Internal Measures 5

2.3.2 External Measures 6

2.4 Object Oriented Software Measures 6

2.5 Frequently Used Software Measures 7

2.6 Characteristics of Software Measures 10

2.7 Software Measure Data Collection 11

2.8 Conventional Approaches for Software Measure Analysis 12

2.8.1 Distribution Analysis 12

2.8.2 Association Analysis 13

2.8.3 Cluster Analysis 14

2.8.4 Summary 15

3 The Self Organizing Maps 16

3.1 Basic SOM 16

3.2 Basic Training Process 16

3.3 Parameter Influence 19

3.3.1 Size of SOM 19

3.3.2 Learning Rate 19

3.3.3 Neighborhood Function 20

3.3.4 Neighborhood Size 20

3.4 Interpretation of Results 21
3.4.1 Weight maps 21
3.4.2 Clustering map 21
3.4.3 Data distribution map 22
3.4.4 Variable Contribution 22
3.5 Performance Measure 23
3.5.1 Accuracy 23
3.5.2 Robustness 23
3.6 SOM Improvement 23
3.6.1 Data Normalization 23
3.6.2 Neuron Visualization 25
3.6.3 Frequency Sensitive 26
3.6.4 Termination Criteria 27
3.6.5 Training Speedup 27
3.6.6 Using Different Distance Functions 30
3.7 Combine SOM with Other Clustering Approach 32
4 Application of SOM to Software Measure Analysis 34
4.1 Comparison with Conventional Analysis 34
4.2 Description of the Experimental Data 36
4.2.1 MIS Data Set 36
4.2.2 Linguist Data Set 37
4.2.3 JDK Data Set 38
4.3 Distribution and Association Analysis 38
4.3.1 Statistics 38
4.3.2 SOM Training 39
4.3.3 Distribution 40
4.3.4 Association 40
4.3.5 Summary 41
4.4 Cluster Analysis 42
4.4.1 Clustering by Single SOM 42
4.4.2 Clustering by Hierarchical SOM 47
5 Using Different Similarity Measures in SOM 59
5.1 A Neural Model for Similarity Measure 59
5.2 Experiment with Synthetic Data 60

5.2.1 Description of Experimental Data 60
5.2.2 Supervised Learning in Neural Networks 61
5.2.3 Unsupervised Learning in SOM 62
5.3 Experiment with Software Measure Data 63
5.3.1 Supervised Learning in Neural Networks 63
5.3.2 Unsupervised Learning in SOM 65
5.4 Summary 66
6 Granular Description of SOM 67
6.1 Motivation 67
6.2 Using Sets as Granular Descriptor of SOM 68
6.3 Using Fuzzy Sets as Granular Descriptor of SOM 69
6.4 Experiment with Synthetic Data 71
6.4.1 SOM for Synthetic Data 71
6.4.2 Set Descriptors of SOM clusters for Synthetic Data 72
6.4.3 Fuzzy Set Descriptors of SOM clusters for Synthetic Data 74
6.4.4 Using Fuzzy Set Union to Describe Larger Clusters in Synthetic Data 75
6.5 Experiment with Software Measure Data 75
6.5.1 SOM for Software Measure Data 75
6.5.2 Set Descriptors of SOM Clusters for Software Measure Data 76
6.5.3 Fuzzy Set Descriptors of SOM clusters for Software Measure Data 78
6.5.4 Using Fuzzy Set Union to Describe Larger Clusters in Software Measure Data 80
6.6 Summary 81
7 SOM Tool Design 82
7.1 Requirement Analysis 82
7.2 User interaction modeling 83
7.3 User interface architecture design 84
7.4 Major object identification 85
7.5 Implementation 85
7.5.1 Platform 86
7.5.2 Programming Language 86
7.5.3 Framework 86
7.5.4 Overall Class Diagram 87
7.6 Testing 89
7.6.1 Handling Large Data Set 89

7.6.2 Computation Speed 89
7.7 Future Improvements 90
7.8 Summary 91
8 Conclusion 92
References 94
Appendix 97
1 SOM Tool Package 97
2 Screen Shots for SOM Tool 98

List of Formulas

(2.1) Calculation of Correlation 14

(2.2) Calculation of covariance 14

(3.1) Determining winning neurons 16

(3.2) Modifying neuron connection weights 17

(3.3) Determining the index of winning neuron 18

(3.4) SOM neighborhood function 20

(3.5) Linear normalization 24

(3.6) Logarithmic normalization 25

(3.7) Logistic normalization(step 1) 25

(3.8) Logistic normalization(step 2) 25

(3.9) Euclidean distance Function 30

(3.10) Tchebychev distance Function 30

(3.11) Hamming distance Function 30

(5.1) Fuzzy exclusive or function 60

(6.1) Set membership function 68

(6.2) Construction of fuzzy membership function 70

List of Tables

Table 3.1 A synthetic data set 24

Table 3.2 The linearly normalized data 24

Table 3.3 The logarithmically normalized data 25

Table 3.4 The logistically normalized data 25

Table 3.5 Required memory to store the N-Table 28

Table 3.6 Noise influence of SOM for the Linguist data using different distance functions 32

Table 4.1 Software measures for the MIS data 36

Table 4.2 Software measures used in The Linguist and the JDK data 37

Table 4.3 Major source code packages provided by JDK 37

Table 4.4 Statistics for the MIS data 39

Table 4.5 Correlation matrix for THE MIS data 39

Table 4.6 The "hot" neurons(highly homogeneous clusters) in Figure 4.5 45

Table 4.7 Some university data 48

Table 4.8 Cluster profiles for the 5 clusters in Figure 4.18 54

Table 4.9 Descriptions of SOM clusters based on the software module names 54

Table 4.10 Cluster prototypes for the 6 clusters in Figure 4.22 56

Table 4.11 The "hot" neurons(highly homogeneous clusters) in Figure 4.23 57

Table 6.1 Set descriptions of SOM clusters for synthetic data using 3 sets 73

Table 6.2 Set descriptions of SOM clusters for synthetic data using 5 sets 73

Table 6.3 Fuzzy Set descriptions of SOM clusters for synthetic data using 3 sets 74

Table 6.4 Fuzzy Set descriptions of SOM clusters for synthetic data using 5 fuzzy sets 75

Table 6.5 Using Fuzzy Set Unions to describe SOM clusters for synthetic data 75

Table 6.6 Set descriptions of SOM clusters for The Linguist software measure data using 3 sets 77

Table 6.7 Set descriptions of SOM clusters for the Linguist software measure data using 5 sets 78

Table 6.8 Fuzzy Set descriptions of SOM clusters for the Linguist software measure data using 3 sets 79

Table 6.9 Fuzzy Set descriptions of SOM clusters for the Linguist software measure data using 5 fuzzy sets 80

Table 6.10 Using Fuzzy Set Unions to describe SOM clusters for The Linguist software measure data 81

Table 7.1 Testing the ability of SOM tool to handle large data set 89

Table 7.2 A comparison of computation time between SOM tool and Matlab code 90

List of Figures

Figure 2.1 Some UML diagram examples 7

Figure 2.2 Deriving DIT and NOC from UML class diagram 8

Figure 2.3 Deriving NOA and NOM from UML diagram 9

Figure 2.4 Box plot with annotations for each part 13

Figure 2.5 Scatter plot for LOC and Changes in THE MIS data set 14

Figure 2.6 Trajectories of FCM clustering 15

Figure 3.1 An artificial model for self organizing neural network 16

Figure 3.2 The training process in self organizing map 17

Figure 3.3 The SOM training process for a simple synthetic data set 18

Figure 3.4 Two learning rate curves 19

Figure 3.5 Different neighborhood functions 20

Figure 3.6 SOM weight matrix and its derived maps 21

Figure 3.7 SOMs with different shapes and neurons 26

Figure 3.8 N-Table(Neighborhood Table) when radius = 1 for a 5x5 SOM 28

Figure 3.9 The SOM fast winner searching by region 29

Figure 3.10 SOM weight maps for The Linguist data set using three different distance functions 31

Figure 3.11 The 3-D synthetic data set 32

Figure 3.12 The SOM clustering map and the FCM cluster prototypes 33

Figure 4.1 The simplified process of collecting software measure data by WebMetrics 36

Figure 4.2 Histograms for all the software measures in THE MIS data 38

Figure 4.3 Data distribution map for THE MIS data 40

Figure 4.4 Weight maps for THE MIS data 41

Figure 4.5 SOM clustering map for The Linguist data 43

Figure 4.6 SOM data distribution map for The Linguist data 43

Figure 4.7 SOM weight maps for all software measures in The Linguist data 44

Figure 4.8 Histograms for all software measures in The Linguist data 45

Figure 4.9 Clustering map and data distribution map for the Linguist data without DIT 46

Figure 4.10 Weight maps for the Linguist data without DIT 46

Figure 4.11 The structure of hierarchical SOM 47

Figure 4.12 SOM for university clustering 48

Figure 4.13 Hierarchical SOM for university clustering 49
Figure 4.14 Clustering map for the JDK data 50
Figure 4.15 Distribution map for the JDK data 50
Figure 4.16 Weight maps for the JDK data 50
Figure 4.17 Histograms of all software measures for all the modules of the JDK data ... 51
Figure 4.18 Histograms of all software measures for the modules in Cluster-B 52
Figure 4.19 Clustering map for the JDK data in Cluster-B 53
Figure 4.20 Data distribution map for the JDK data in Cluster-B 53
Figure 4.21 Weight maps for the JDK data in Cluster-B 53
Figure 4.22 Histograms of all software measures for the modules in Cluster-B-5 55
Figure 4.23 Clustering map for the JDK data in Cluster-B-5 56
Figure 4.24 Data distribution map for the JDK data in Cluster-B-5 56
Figure 4.25 Weight maps for the JDK data in Cluster-B-5 57
Figure 4.26 Building hierarchical SOM 58
Figure 5.1 SOM training architecture based on neural network similarity measure 59
Figure 5.2 Basic network structure for the training of neural similarity measure 60
Figure 5.3 Scatter plot of distances in the input space and output space 60
Figure 5.4 Performance index Q in successive learning epochs 61
Figure 5.5 Scatter plot of similarity measures in the input and output space 61
Figure 5.6 Visualization of fuzzy XOR data on the SOM constructed using the neural network similarity measure 62
Figure 5.7 Visualization of fuzzy XOR data on the SOM using Euclidean distance 62
Figure 5.8 Performance index Q in successive learning epochs 64
Figure 5.9 Correspondence between neural similarity measure and difference of "Changes"(Normalized) for 2500 MIS data pairs 64
Figure 5.10 Correspondence between Euclidean distance and difference of "Changes"(Normalized)) for 2500 MIS data pairs 64
Figure 5.11 The SOM using neural similarity measure for 50 MIS data records 65
Figure 5.12 The SOM using Euclidean distance for 50 MIS data points 65
Figure 6.1 Comparison between granular description and numerical description for a cluster in SOM 67

Figure 6.2 Sets constructions for software measures LOC and NOM 68
Figure 6.3 Construction of Sets for a software measure data set 68
Figure 6.4 Construction of fuzzy sets for one variable in a software measure data set 70
Figure 6.5 2-D synthetic data set 71
Figure 6.6 Self organizing maps for a 2-D synthetic data set 72
Figure 6.7 Sets construction for the synthetic data set 72
Figure 6.8 Visualization of set descriptors for the synthetic data 73
Figure 6.9 Sets construction for the synthetic data set 73
Figure 6.10 Fuzzy Sets construction for the synthetic data set 74
Figure 6.11 Fuzzy Sets construction for the synthetic data set 74
Figure 6.12 Self organizing maps for The Linguist software measure data set 76
Figure 6.13 3-set constructions for the Linguist software measure data set 77
Figure 6.14 5-set constructions for the Linguist software measure data set 78
Figure 6.15 3-fuzzy-set constructions for the Linguist software measure data set 79
Figure 6.16 5-fuzzy-set constructions for the Linguist software measure data set 80
Figure 7.1 State diagram for user interaction modeling 83
Figure 7.2 Simplified screen flow chart for the SOM Tool 84
Figure 7.3 Screen partition for result displaying and cluster analysis 84
Figure 7.4 UML Class diagram for SOM tool implementation 88
Figure 7.5 The training time for different size of data set using SOM Tool 90
Figure A.1 Startup Screen 98
Figure A.2 Open Existing File Screen 98
Figure A.3 Training Data Setting Screen 98
Figure A.4 SOM and Training Parameter Setting Screen 99
Figure A.5 Map Displaying Setting Screen 99
Figure A.6 Main screen for SOM analysis 100
Figure A.7 Cluster Analysis Menu 100
Figure A.8 List the Raw Data from Selected Cluster 101
Figure A.9 Grow Child SOM from Selected Cluster in Parent SOM 101

1 Introduction

The fast growing software industry requires precise control and efficient management of software development. Software measurements capture the quantitative characteristics of software and enable quantitative software engineering. By analyzing software measures, software managers and developers can get thorough insights into software projects. The conventional approach to analyze software measures is by statistical means. However, software measures do not always satisfy the basic statistical assumptions such as normality and constant variance. Therefore, we introduce self organizing maps, a particular neural network approach, to analyze software measures. By visualizing highly-dimensional data on 2-dimensional maps, SOM provides us with a user-friendly and interactive vehicle to analyze software measures.

1.1 Software Measurements

In the software industry, the engineering approach plays a more and more important role. The engineering approach means that each activity is understood and controlled, so that there are few surprises as the software is specified, designed, built, and maintained(Fenton, 1997).

Software measurements can help us understand and control software development. There are some measures that help us to understand what is happening during development and maintenance. The measurements make the aspects of process and product more visible to us, giving us a better understanding of relationships among activities and the entities they affect. Software measurements allow us to control what is happening on our projects. Using our baselines, goals and understanding of relationships, we predict what is likely to happen and make changes to processes and products that help us to meet our goals. For example, we may monitor the complexity of code modules, giving thorough review only to those that exceed acceptable bounds. Software measurements also encourage us the improve our software processes and products. Applying software measurements, the complexity and quality of software can be compared and ranked, the software processes can be precisely evaluated. Thus we can determine how well we have done and make further improvements in the right direction.

1.2 Software Measure Analysis

Software measures have some characteristics that make them difficult for analysis. First, they are highly-dimensional, because software is such a complex entity that just a few measures cannot completely capture its properties. Second, there is a high level of uncertainty between internal software properties and external software behavior, so it is very difficult to build precise statistical models for software measures. Finally, software measures are highly associative. Software modules can have many software measure variables, but few of them are independent variables. This characteristic adds more difficulties to software measure analysis(Fenton, 1997).

To understand and control software development, we usually apply statistical means to analyze software measures. Statistics arose originally as a response to intensive experimentation in physical environments, which adhere quite well to fundamental assumptions, such as normality, constant variance, and continuity of the phenomena. Both software engineering products and processes are far more abstract than physical environments. This is the main reason for the experimental data to deviate quite radically from any standard assumptions as to the underlying probability distributions as well as the continuity principle. For example, small changes to the requirements may result in far drastic and radical changes in the total cost of the overall software project (Pedrycz, 2001a).

Neural-computing has been proven to be an effective means to capture and represent the structure of experimental data. In the problems of analysis of software measures, neural networks are attractive because of their learning abilities and inherently nonlinear characteristics(Pedrycz, 2001a). Using neural networks to analyze software measures, some statistical assumptions can be relaxed.

1.3 Self Organizing Maps

Kohonen's Self-Organizing Maps(SOM) is one of the most popular artificial neural network algorithms(Kohonen, 1995). It can project highly-dimensional patterns onto a low dimensional(usually two dimensional) space in such a way that a topology of the data is preserved. The basic Self-Organizing Map can be visualized as a sheet-like neural-network matrix. The cells (or nodes) become specifically tuned to various input signal patterns or classes of patterns in an orderly fashion. The learning process is competitive and unsupervised, meaning that no teacher is required to define the correct output for an input. SOM has the ability to preserve the topology of highly-dimensional data, so if we apply SOM to software measure analysis, it will tell us graphically what the data looks like and how it is interrelated.

Compared with conventional statistical software measure analysis approaches, the most obvious advantage of SOM analysis is its potential as a visualization tool. A trained SOM can convey a large amount of information graphically(Kaski, 1997), while statistical numbers and diagrams convey far less information. In some sense, the learned SOM can be viewed as a condensed data image. Nearly all the information about the original data is preserved in a collection of maps.

1.4 Research Objectives

The general objective of this thesis is to provide software managers and developers with a vehicle for analyzing software measure data to easily disclose the relationship and clustering of software modules. To be specific, we have the following research objectives:

- Comprehensive introduction of software measurements
  We will discuss what software measurements are, why they are important to software development and how to collect software measures. We will introduce some frequently used software measures, especially the object oriented software measures. Some conventional approaches for software measure analysis are also introduced.

- In-depth investigation of SOM algorithms
  The SOM algorithm is discussed in detail including basic algorithms, parameter influences, interpretation of results and performance measures. Some improvements is made to the basic SOM algorithms.

- Experimentation of applying SOM in software measure analysis
  The SOM analysis is applied to software measure data from real software projects. The advantages of SOM analysis to statistical analysis is discussed. The user-friendly and interactive features of SOM in software measure analysis is demonstrated, including graphical analysis on data distribution, graphical analysis on feature relationships and interactive cluster visualization.

- Extensions of SOM in software measure analysis
  In order to make SOM fit better in software measure analysis, some extensions is made to SOM. First, we will add partial supervision to SOM by combining supervised learning and unsupervised learning. This is achieved by applying a neural model to measure similarity

between software modules. Second, we will apply granular descriptors to describe SOM clusters, so that SOM will become a more user-friendly tool to analyze software measures.

- Building a software tool for SOM analysis
  To make SOM become a practical tool for users to analyze software measures, a software package is developed. This software will implement the SOM algorithms, graphical display, friendly user interface and experimental data management.

1.5 Thesis Organization

The material is arranged into eight chapters. Chapter 2 discusses the concepts, motivation, classification and characteristics of software measurements. Some frequently used software measures and conventional approaches to analyze them are also introduced in Chapter 2. Chapter 3 focuses on Self Organizing Maps. The basic algorithms and some improvement is discussed there. We also compare SOM with other clustering approaches, such as FCM in Chapter 3. In Chapter 4, we discuss the application of SOM in software measure analysis. First, a comparison between SOM and conventional analysis is presented. Then, we use SOM to analyze some real software measure data sets to demonstrate the uniqueness and effectiveness of SOM as a data-mining vehicle. In Chapter 5, a neural network based similarity measure is introduced. We can see how this neural model can improve the performance of SOM. In Chapter 6, we discuss the granular description of SOM clusters. Sets and fuzzy sets is applied to describe SOM clusters in a way that is objective and easy to understand. In Chapter 7, in order to make practical use of SOM in software measure analysis, an interactive SOM tool is discussed in detail. Finally, Chapter 8 makes a brief conclusion of this thesis. In the appendix, the SOM Tool package is introduced and some screen-shots are presented.

2 Software Measurements

Because of the fast growth of the computer industry, software development becomes more and more complex. In order to deal with the complexity of software projects, we need to apply engineering principles to manage software development. Software measurements are the first step to apply quantitative software engineering in software development. By analyzing and managing the software measures, we can control the software cost, complexity and quality.

2.1 General Concept of Measurements

Measurements are the processes by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules(Fenton, 1997). In this definition, entities refer to the real objects such as dogs and books. The attributes refer to the properties of those real world objects, such as the weight of a dog and the pages of a book. The nature of measurements is a mapping from real world to the numerical system, from our empirical concept to numerical system. Therefore software measurements are also mappings from software attributes to the numerical system which will properly capture the properties of software.

2.2 Motivation of Software Measurements

Software measurements are motivated by the increasing complexity of software development. We can not control what we can not measure(Demarco, 1982). In order to control the large, complex software development process, we must first get the measures of software.

2.2.1 Fast Growing Complexity of Software

Several decades ago, people wrote software with hundreds to thousands of lines of code. The complexity of software was manageable by programmers without using any tools. Nowadays, a middle sized commercial software project may have tens of thousands of lines of code. A large project may easily exceed millions of lines of code. For example, a Medical Imaging System(Munson, 1996) has 40,000 modules and over 400,000 lines of code. For such a huge software project, it is impossible for people to manage and control it just by common-sense and intuition. Therefore, people developed quantitative engineering approaches to manage and control large software development.

2.2.2 Software Engineering

Software engineering is a collection of techniques that apply engineering approaches to the construction and support of software products. Software engineering includes such activities as managing, evaluating, planning, modeling, analyzing, specifying, designing, implementing, testing and maintaining(Fenton, 1997). Each of the above activities is based on models and theories, which are usually expressed as formulas and laws. So in order to apply engineering approaches to manage and control software development, we need to know the quantitative characteristics of software. Software measurements can help us acquire the numerical characteristics of software.

2.2.3 Application of Software Measurements

In quantitative software engineering, software measurements play an essential role. Basically, software measurements can be applied to estimate software cost and control the quality of code.

- Development Cost Estimation
  Cost is an important factor for software managers. The software manufacturing processes include requirement analysis, system specification, system design, coding, testing and maintaining. We want to know the cost for each process, so that we can control the budget. Applying software measurements, we may use some models to estimate the cost of those processes in their early lifecycle(MacDonell, 1997).

- Evaluation of Code Quality
  During the software development, if we record all the faults, failures and modifications to the code, we can measure the quality of the code. If the code is properly designed, carefully written and well documented, there are fewer faults, failures and modifications. So we rank this code as high quality code. Once we have the quality measure for all the code modules, we can identify the high quality modules and low quality modules, and eventually find out what cause the quality to vary.

2.3 Classification of Software Measures

In order to describe the various aspects of software, people developed a large number of software measures. These software measures can be classified into different categories according to different rules. Generally, we can identify a software measure as either internal software measure or external measure.

2.3.1 Internal Measures

Internal measures refer to the internal attributes of software. They can be measured purely in terms of the software itself. We can measure them by examine the software internal characteristics without having to observe its behavior. The following is a list of frequently used internal software measures.

- Size
  This is the most frequently used software measure and maybe the most useful one. It can be measured by many software internal attributes. The simplest size measure is lines of code. It has some variation such as lines of code without comments or lines of code without blank lines. Also, we can use other software internal attributes as a size measure such as the number of function points, the bytes of code file and the total characters in the code file.

- Structure
  This internal attribute is very different from the size of software. Structure of software measures the topology of control flow and the hierarchy of objects. Usually a software with more complex structure has larger size. However sometimes this is not true. For example, a code module to implement a sorting algorithm may have less than 100 lines of code, but it is intuitively much more complex than a code module with 300 lines of code but just to output text. A frequently used structure measure is the cyclomatic number which counts the branches and loops in software.

- Fault
  Fault is the mistake that the programmer made during software development. It may be caused by the carelessness of programmer or by the high complexity of software. Fault can be measured by the bugs found or changes made. Fault is associated with the software size, so we often use Fault Rate to measure the quality of software. Fault Rate is the total faults divided by the software size, so that we can compare the fault rate with different size of software.

- Complexity
  Complexity of software is determined by multiple factors. Size and structure are the two most important factors. Other factors such as programming language may also influence the complexity of software. The measure of complexity is very useful in estimating effort and cost for software.

2.3.2 External Measures

External measures refer to the external attributes of software. They are expressed only by interacting with their environment. The behavior of software is much more important for external measures than for internal measures. The following is a list of frequently used external software measures.

- Reliability
  Reliability can be measured by the failure rate. This is an external software attribute, so we must consider the environment in which the software is used. Reliability depends heavily on usage of the software. For example, when we use a word processor to edit a simple text, the reliability of this word processor may be very high. However, if we use this word processor to edit a large complex document, many problems might occur and its reliability might be low.

- Maintainability
  Maintainability means how well and how easy a software can be maintained. Obviously, it is an external software attribute. The software design and documentation can affect the maintainability, but the skills of the maintainers will also affect the maintainability.

- Efficiency
  Efficiency can be measured by the time and memory used to run the software. An efficient software will take shorter time and consume less memory. However, efficiency is also an external software measure and depends on the hardware which the software runs. If we run efficient software on a very slow machine, it may not demonstrate its efficiency.

- Reusability
  A well designed software module can be reused easily. Again, this is an external measure, and it depends on its environment. For example, a software module which is reusable in one operating system may not be reusable in another operating system.

2.4 Object Oriented Software Measures

The goal of software measurements is to predict effort, cost and quality in the early stage of software development. In the early stage of software development, we do not know the lines of code or cyclomatic number of software modules. What we know in the early stage of software development is the architecture and the skeleton of software modules. Currently, most software is developed using OOD(Object Oriented Development) methods. In OOD, people must specify the skeletons for all the classes in the project. For each class, the member variables and member functions must be declared in the early stage. In OOD, the class interaction and relationship must also be specified before implementation. This means that we can get some object oriented software measures in the early stage of software development. Then after analysis on these measures, we may predict the software development effort, cost and quality(Briand, 1999). In the next section, we introduce some frequently used OO(Object Oriented) software measures.

In the early stage of OOD, people usually use Unified Modeling Language(UML) to model the structure and behavior of software projects. UML is a graphical modeling language. It includes class diagrams, state diagrams, collaboration diagrams, use case diagrams, etc(Boggs, 1999). Figure 2.1 shows some sample UML diagrams. We will show some examples on how to derive OO software measures from UML diagrams in the next section.

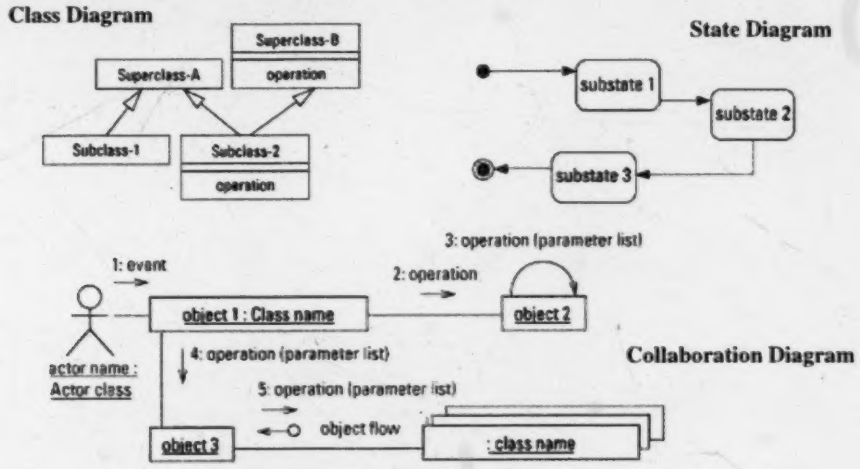


Figure 2.1 Some UML diagram examples

2.5 Frequently Used Software Measures

Because of the diversity and complexity of software objects, there are a large number of measures available to measure software properties. In practice, most software is described by only a few frequently used software measures. In this section, we introduce some frequently used software measures which is referenced in later chapters.

- LOC(Lines Of Code)
  This is the simplest of all the measures to define and calculate. Counting lines of code has a long history as a software measure dating from before the rise of structured programming, and it is still in widespread use today. The size of a software module affects the ease with which it can be understood, its reusability and its maintainability. There are a variety of ways that the LOC can be calculated. These include counting all the lines of code, the number of statements, the blank lines of code, the lines of commentary, and the lines consisting only of syntax such as block delimiters. LOC can be configured to count only executable code, so that all the declaration will not be counted.

- V(g)(Cyclomatic Number)
  This measure was introduced in the 1970s to measure the amount of control flow complexity or branching complexity in a module such as a subroutine(McCabe, 1976). It gives the number of paths that may be taken through the code, and was initially developed to give some measure of the cost of producing a test case for the module by executing each path. Methods with a high cyclomatic complexity tend to be more difficult to understand and maintain. In general the more complex the modules of an application, the more difficult it is to test it, and this will adversely affect its reliability.

LOC and V(g) are general purpose software measures. They can be used to measure both OO(Object Oriented) and non-OO software. In the following section, we will discuss some OO software measures. Unlike LOC and V(g) which have to be measured only after the complete code is written, OO software measures can be measured in the early stage of software development(see discussion in section 2.4).

- DIT(Depth of Inheritance Tree)
  The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree and is measured by the number of ancestor classes(Chidamber, 1994). The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but provide more potential for reuse of inherited methods. Figure 2.2 shows an example to derive DIT directly from UML class diagram.

- NOC(Number Of Children)
  The number of children is the number of immediate subclasses subordinate to a class in the hierarchy(Chidamber, 1994). It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and misuse of subclassing. On the other hand, the greater the number of children, the greater the reuse since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. NOC, therefore, primarily evaluates testability and design. Figure 2.2 shows an example how to derive NOC directly from UML class diagram.

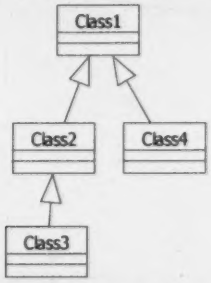


Figure 2.2 Deriving DIT and NOC from UML class diagram
DIT(Class1) = 1, NOC(Class1) = 2
DIT(Class2) = 2, NOC(Class2) = 1
DIT(Class3) = 3, NOC(Class3) = 0
DIT(Class4) = 2, NOC(Class4) = 0

- NOA(Number of Attributes)
  The number of distinct state variables(attributes) in a class serves as one measure of its complexity. The more state a class represents the more difficult it is to maintain invariant for it. It also hinders comprehensibility and reuse. In OO language such as Java, state can be exposed to subclasses through protected fields, which entails that the subclass also be aware of and maintain any invariant. This interference with the class's data encapsulation can be a source of defects and hidden dependencies between the state variables. NOA counts the number of fields declared in the class. Figure 2.3 shows an example to derive NOA directly from UML class diagram.

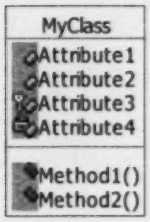


Figure 2.3 Deriving NOA and NOM from UML class diagram
NOA(MyClass) = 4
NOM(MyClass) = 2

- NOM(Number of Methods)
  NOM is also called NLM(Number of Local Methods). It serves best as an interface measure. NOM indicates the operation property of a class. The more methods a class has, the more complex the class's interface(Li, 1993). NOM can be configured to include the local methods of all of the class's super-classes. Methods with public, protected, package and private visibility can be independently counted by using different configurations. Figure 2.3 shows an example to derive NOM directly from UML class diagram.

- MPC(Message Passing Coupling)
  MPC is used to measure the complexity of message passing among classes. Since the pattern of the message is defined by a class and used by objects of the class, the MPC measure also gives an indication of how many messages are passed among objects of the classes. The MPC value is counted by the number of "send" statements defined in a class. The number of messages sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes(Li, 1993).

- WMC(Weighted Methods per Class)
  WMC measures usability and reusability(Li, 1993). It is the count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement since not all methods are assessable within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children since children will inherit all the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

- RFC(Response For Class)
  This measure evaluates system design as well as the usability and the testability. The RFC is the cardinality of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class(Chidamber, 1994). This includes all methods accessible within the class hierarchy. This measure looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. The largest possible value of RFC will assist in the appropriate allocation of testing time.

- CBO(Coupling Between Objects)
  CBO evaluates design implementation and reusability. It is a count of the number of other classes to which a class is coupled(Chidamber, 1994). It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier to reuse it in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation.

- LCOM(Lack of Cohesion Of Methods)
  This measure evaluates the design implementation as well as reusability. LCOM measures the degree of similarity of methods by instance variables or attributes(Chidamber, 1994). Any measure of separateness of methods helps identify flaws in the design of classes. The LCOM can be derived by the following steps.
- Calculate for each data field in a class what percentage of the methods use that data field.
- Average the percentages then subtract from 100%.
- Lower percentages mean greater cohesion of data and methods in the class.
  High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion.

2.6 Characteristics of Software Measures

Compared with the measures of other objects, software measures are more complex. The following is a list of some software measure characteristics which make software measures difficult for analysis.

- High Dimensionality
  Software has to be measured from different aspects. For each code module, there are dozens of measures to describe it, such as LOC, DIT and NOC. These measures form a highly-dimensional data set. This characteristic increases the analysis complexity and computational cost.

- Different variables are highly interrelated
  Software measures generate highly-dimensional data. Each variable is not independent of other variables. Instead, many software measures are closely interrelated. For example, LOC is closely interrelated to V(g) because a long code module tends to have a large cyclomatic number.

- Broad range of values
  In software measure data set, each data record usually represents a software code module(or class). In a software project, code modules are very heterogeneous. Large modules may have thousands of lines of code, while small modules only have a few lines of code. So the data range is very broad. Another characteristic that makes the software measure data difficult to analyze is the data distribution. The software measure data are usually skewed to the low end. For example, in the MIS data set(Munson, 1996), most software modules have small LOC and very few modules have large LOC.

- Uncertainty of relationships with external software behavior
  A major goal of quantitative software engineering is to predict external software behavior by internal software measures. The software measure values that we get from software source code are precise. However, their relationship to the external software behavior is not precise. Unlike the measures in chemistry and physics, in which we can build accurate models on them, software entities have too much uncertainty to build accurate models between internal measures and external behavior. For example, the effort to develop software is usually measured by the size of the code. However, in many cases, a small algorithm code module may be very difficult to write and maintain, while a large initialization code module may be very straight forward and easy to write and maintain. Therefore, the small algorithm module will require more effort than the large initialization module.

- Different measurement standards
  Software is a complex entity. Different types of software have to satisfy different standards. For example, size is the most frequently measured software aspect. Usually, software size can be measured by LOC. However, some software is written by visual programming tools, such as LabVIEW(Bishop, 1999). This kind of software can not be measured in terms of lines of code. They have to be measured by some other measures that need to be related to the symbols and icons in the graph.

2.7 Software Measure Data Collection

So far we have discussed software measures and provided the meaning of many frequently used software measures. Then the next step is to collect data from software projects using software measures. However, since there are too many software measures, it is impossible to collect all the measures from a software project. Therefore we need to choose the suitable software measures for a certain software project.

Software projects are quite diverse in terms of their function, implementation language and complexity. It is very difficult to know in advance which software measures are suitable for a particular project. However we can use some general purpose software measures for software projects. For example, LOC is a generic software measure and can be applied in any software project whose source code is written in text. Once we include LOC as a software measure, we may exclude some software measures that are directly interrelated with LOC, such as TChar(Total number of Characters). In OO(Object Oriented) software, we can use those OO software measures, such as DIT and CBO.

However, for a particular software project, we must choose suitable software measures based on experiment. First, we should follow the general rules to collect data using the frequently used software measures. Then we examine the data we collected, and determine if they really make sense for this software project. Sometimes, we may find that many software measures are closely interrelated in a project. This implies that we are repeatedly measuring the same aspect of the software modules in this project. Therefore, these redundant software measures do not make too much sense. Sometimes we may find that the collected data for a software project is skewed to the low end. That is, the majority of data records only have small values and only very few data records have high values. It makes these high value records look like outliers or noise. In fact, these high value data records are not noise, but real data from some particular software modules. In data analysis, these "outliers" will make the analysis complicated and deteriorated. In many analysis approaches, these "outliers" are simply eliminated as noise. Figure 4.8 illustrates this situation. In Figure 4.8, all the software measures except DIT are skewed to the left. These measures depend on the size of the software modules. Because the majority of modules in this

project are small modules, it is inevitable to see a skewed data set. On the other hand, DIT is independent of the module size, so it has much flatter histogram than other measures.

This problem indicates that the software measures we chose for this project may not be appropriate, or at least not sufficient. Some measures must have been missed to describe those "outliers". The noise situation is caused not by the software itself, but by the lack of software measures. In our daily life, we can also encounter this situation. For example, we want to collect some physical data from a group of people. This group consists of a large number of children and a few adults. If we only collect such measures as weight, height and strength, then we will get a skewed data set. The data from adults will look like noise and will make the data analysis difficult. On the other hand, we can apply some other measures to this group of people, such as skin color, blood type and birth date. The values of these variables will distribute evenly to all the people in this group. These measures make more sense for this group than the measures that generate skewed data. In quantitative software engineering, when collecting data for a software project, we must try to find some measures that can make sense for all the modules in the project. In Figure 4.8, most software measures are interrelated with the size of software modules, so we need to find some measures that is independent of size, such as DIT.

2.8 Conventional Approaches for Software Measure Analysis

The conventional approach to analyze software measures is by statistical means. Usually, people apply the following statistical approach to analyze software measures(MacDonell, 1994).

2.8.1 Distribution Analysis

As discussed from section 2.7, we must know the distribution of the software measure data set before any analysis. Once we know the data distribution information, we can determine if we have applied appropriate software measures, and determine the corresponding statistical formulas. The following is a list of some frequently used statistical indicators to describe data distribution.

- Range of values
  The first step to analyze software measures is to determine the range of data. Software measures have very broad ranges. Normally, we use maximum and minimum values to indicate the range of software measure values.

- Mean and Median
  Once we know the range of measure values, we also want to know the average value of software measures. We can use mean or median to determine the average value of software measures. When data are evenly distributed, mean is good for determining average. However, when data are skewed, as the software measures data usually are, mean will not give us a good indication of average. In this case, we use median. Median is the value in the middle of a ranked array. So if data are skewed to the low end, the middle ranked value will also be close to the low end. So the median represents the average better than mean if data is not evenly distributed.

- Standard Deviation
  In order to know the variation of a software measure, we use standard deviation. It indicates how much the data deviate from the average. In software measure data, most measures have high standard deviations, because they have broad ranges. However, in OO measures, there are a few measures with small standard deviations, such as DIT.

- Histogram
  In order to know the distribution of software measure values, we use histograms. A histogram uses a number of bins representing many sub-ranges. The height of each bin represents the count of data that is in this sub-range. If data are evenly distributed, the histogram is flat. If the data are distributed mostly in the low end, then the histogram will have high bins in the low end, and low bins in the high end. Histogram is a very straight forward graphical approach to show the data distribution.

- Box plots
  Box plot is derived from some statistical indicators: median, upper quartile and lower quartile. We can use box plot to visualize the data distribution. Figure 2.4 shows how to construct a box plot(Fenton, 1997).

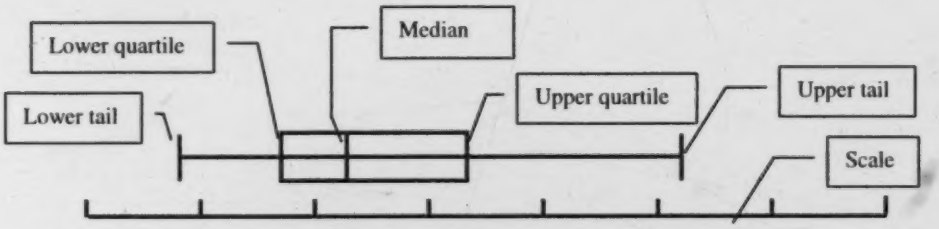


Figure 2.4 Box plot with annotations for each part

The scale is between minimum value and the maximum value. The lower quartile is the median of the data which are smaller than Median. The upper quartile is the median of the data which are larger than Median.
The box length d = Upper quartile - Lower quartile
The Upper tail = Upper quartile + 1.5 * d
The Lower tail = Lower quartile - 1.5 * d
These values should be truncated to the nearest actual data point to avoid meaningless values(out of scale values).

2.8.2 Association Analysis

Association means how closely data variables are interrelated. In software measure analysis, we want to find if there is strong association between two software measures in a particular software project. For example, in the MIS(Munson 1996) data set, we found that LOC(Lines Of Code) is strongly interrelated with TChar(Total Characters). This strong association tells us that in this MIS project, we can use LOC to indicate TChar, or vice versa. It also implies that there is information redundancy in this data set. If we need to reduce the size of data set, we can remove the redundant software measures. In conventional software measure analysis, people often use correlation and scatter plot to show the data association.

- Correlation
  In order to determine the association between different software measures, we use correlation analysis. If two software measures have high correlation, then they are closely interrelated. If two software measures have low correlation, then they are almost independent of each other. Because software measure data are highly dimensional, we must know all the correlation between any two variables. We can use a correlation matrix to store the complete associations of a software measure data set. In section 4.3.1, there is an example of correlation matrix (Table 4.5). The formula to calculate correlation between two variables x and y is shown below:

$$Correlation = \frac{\mathrm{cov}(x, y)}{std(x) * std(y)} \tag{2.1}$$

$$\mathrm{cov}(x, y) = \frac{\sum_{i=1}^{N}[(x_i - mean(x))(y_i - mean(y))]}{N} \tag{2.2}$$

In the above formula, x and y are two variables with the same number of observations N. *mean* is the mean value of a variable. *std* is the standard deviation of a variable. cov(x, y) is the covariance between two variables x and y.

- Scatter plots
  In order to show the association between two software measures, we can draw scatter plots for two measures. Scatter plot shows the relationship of two software measures in a very straight forward way. The data is plotted in 2 dimensions and we can then visualize the association directly. However, if there are too many data points in a software measure data set, drawing a scatter plot may become difficult. Because scatter plot can only show the relationship between two measures, if we want to know the relationships of all the software measures, we have to draw a large number of scatter plots. Figure 2.5 shows a scatter plot for LOC and Changes in the MIS data set(Munson, 1996).

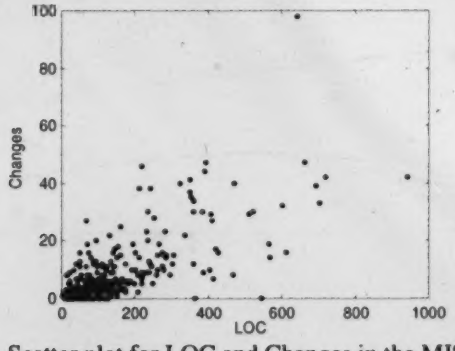


Figure 2.5 Scatter plot for LOC and Changes in the MIS data set

2.8.3 Cluster Analysis

Cluster analysis plays an important role in software engineering. In a large project, with thousands of modules, it is impossible to manage and maintain these modules in one group. Software managers may group similar modules together to make many different module groups and then dedicate different software developers to different groups. In this way, developers can concentrate on a specific problem in a specific field, and will have higher efficiency and productivity. In addition, because similar software modules tend to exhibit similar behavior, once we find a design defect in one module, we may easily go to other modules in the same group and fix the problem before failure occurs.

In conventional software measure analysis, FCM(Fuzzy C-Means) is frequently used as a clustering approach. In FCM, each data point belongs to a cluster to some degree that is specified by a membership grade. FCM provides a method of how to group data points that populate some multidimensional space into a specific number of different clusters. The training of FCM starts with an initial guess for the cluster centers, which are intended to mark the mean location of each cluster. It also assigns every data point a membership grade for each cluster. The initial guess for these cluster centers is most likely incorrect. By iteratively updating the cluster centers and the

membership grades for each data point, the training process iteratively moves the cluster centers to the "right" location within a data set. This iteration is based on minimizing an objective function that represents the distance from any given data point to a cluster center weighted by that data point's membership grade(Bezdek, 1981).

Figure 2.6 shows a synthetic data set and the trajectories of FCM clustering. Once we specify the number of clusters, the FCM can be trained quickly. The cluster centers(prototypes) and the training trajectories are shown on the original data plot. The training trajectories refer to the paths along which cluster centers move during the training. They clearly show how the training proceeds.

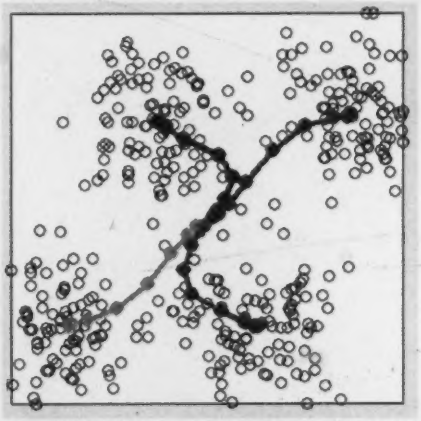


Figure 2.6 Trajectories of FCM clustering(a snapshot from Matlab Toolbox demo) There are 4 clusters in this synthetic data set. The trajectories show how the FCM learning converges to the cluster centers.

The disadvantage of FCM is that we must know the number of clusters in advance. The FCM algorithm can not determine how many are clusters actually in the data set. We have to use an adhoc method to find the best guess. In addition, the FCM algorithm only gives us some numbers. We cannot see the entire clustering geography. It is difficult to know what the boundaries between clusters look like.

2.8.4 Summary

We have discussed some statistical approaches for analyzing software measures. They are suitable for simple analysis. If we want to analyze and predict more complex variables, such as effort and complexity, these simple statistical approaches may not do the job. For multi-criteria, multivariate data analysis, some more sophisticated approaches is used, such as neural networks. In the following chapters, we discuss a particular neural network approach, Self Organizing Maps, to analyze software measure data. We can see how this powerful data-mining tool applies to software measure analysis.

3 The Self Organizing Maps

The self organizing map is a method for producing ordered low-dimensional representations of an input data space. The input data is typically complex and highly-dimensional with data elements being related to each other in a nonlinear fashion. SOM can project the highly-dimensional input data space onto a low-dimensional space(usually 2-D or 3-D) in such a way that a topology of the original data is preserved. Once we get the low-dimensional representation of input data space, we can easily visualize the data clusters and relations of variables.

3.1 Basic SOM

Self Organizing Maps(SOM) is an artificial self organizing neural network that emulates the self organizing structure in the human brain. SOM simplifies the human neural network models and preserves the major properties of real neural systems. Figure 3.1 shows the artificial model of neurons and network. The network consists of a 2-D matrix of neurons. The location of each neuron is fixed. Each neuron is identified by its location and is associated with a weight vector. These weight values represent the neural connection strengths in biological systems. The output activity of the neuron is simply the modification of its weight vector in response to the input vectors. In SOM, neurons are laterally interconnected. Each neuron is connected to a subset of its neighbors. Any output in one neuron will generate output to its neighboring neurons. To model the biological system, the output of these neighboring neurons will decline further away from the activation neuron(Flanagan, 1996 and Kiviluoto, 1996).

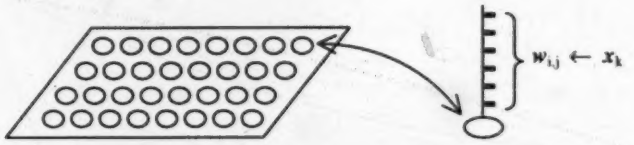


Figure 3.1 An artificial model for self organizing neural networks
The network consists of a 2-D array of neurons. Each neuron has a weight vector $\boldsymbol{w}_{i,j}$, where (i, j) denotes the coordinates of this neuron on the map. The input pattern $\boldsymbol{x}_k$ is compared with $\boldsymbol{w}_{i,j}$ for all neurons to find the winning neuron.

3.2 Basic Training Process

First let's denote $\boldsymbol{w}_{i,j}$ *as* the weight vector of the neuron with coordinates (i, j), where i =1,2...COL, j = 1,2...ROW, COL denotes the total number of columns in the map and ROW denotes the total number of rows in the map. $w_{i0,j0}$ denotes the weight vector of the winning neuron with coordinates (i_0, j_0) on the map. It is important to note that the weight vectors have the same dimensionality as the input patterns. The training process of self-organizing maps may be described in terms of input pattern presentation and weight vector adaptation. Each training iteration t starts with the random selection of one input pattern $\boldsymbol{x}(t)$. This input pattern is presented to the self-organizing map and each neuron in the map is compared with this input pattern. Usually, the Euclidean distance is used to calculate the distance between the weight vector of a neuron and the input pattern. However, other distance functions can also be used, such as Hamming distance and Tchebychev distance. The neuron with the smallest distance is referred to as the winning neuron c. Formula 3.1 shows how to select the winning neuron.

$$\left\|\mathbf{x}(t) - \mathbf{w}_{i_0,j_0}(t)\right\| = \min_{i,j}\left\|\mathbf{x}(t) - \mathbf{w}_{i,j}(t)\right\| \qquad (3.1)$$

In Formula 3.1, $\|\boldsymbol{a}\text{-}\boldsymbol{b}\|$ denotes the distance between vector $\boldsymbol{a}$ and vector $\boldsymbol{b}$. The winning neuron $\boldsymbol{w}_{i0,j0}$ is the closest to the input data record $\boldsymbol{x}$. Finally, the weight vector of the winner(winning

neuron) as well as the weight vectors of selected neurons in the vicinity of the winner are adapted. This adaptation is implemented as a gradual reduction in the difference between an input pattern and an weight vector, as shown in Formula 3.2.

$$\mathbf{w}_{i,j}(t+1) = \mathbf{w}_{i,j}(t) + \alpha(t) * N(i, j, i_0, j_0, t) * [\mathbf{x}(t) - \mathbf{w}_{i,j}(t)] \qquad (3.2)$$

Geometrically speaking, the weight vectors of the adapted neurons are moved gradually towards the input pattern. The amount of weight vector movement is guided by the learning rate, α, which will decrease during training. The strength of adaptation in the neighborhood neurons is determined by the neighborhood function, $N(i, j, i_0, j_0, t)$. This function is usually a discrete Gaussian function. The closer a neuron is to the winner at (i_0, j_0), the higher the strength of adaptation will be for this neuron. The range of this function also decreases over training. The consequence of this process is that the distance between the input pattern and the weight vectors decreases. Thus the weight vectors become more similar to the input patterns. The winning neuron is more likely to win at future presentations of this input pattern. The consequence of adapting for both the winner and its neighborhood neurons leads to a spatial clustering of similar input patters in neighboring parts of the self organizing maps. Therefore, the topology in the input patterns that are present in the n-dimensional input space is projected to the two-dimensional output space of the self-organizing map. The training process of the self-organizing map describes a topology preserving mapping from a highly-dimensional input space onto a two-dimensional output space where patterns that are similar in terms of the input space are mapped to geographically close locations in the output space.

Consider Figure 3.2 for a graphical representation of self organizing maps. The map consists of a square arrangement of neural processing elements(neurons), shown as circles on the left-hand side of the figure. The black circle indicates the neuron that was selected as the winner for the presentation of input pattern $\boldsymbol{x}(t)$. The weight vector of the winner, $\boldsymbol{w}_{i0,j0}(t)$, is moved towards the input pattern and thus, $\boldsymbol{w}_{i0,j0}(t+1)$ is nearer to $\boldsymbol{x}(t)$ than was $\boldsymbol{w}_{i0,j0}(t)$. Similarly, adaptation is performed with a number of neurons in the vicinity of the winner. These neurons are marked as shaded circles in Figure 3.2. The degree of shading corresponds to the strength of adaptation. Thus, the weight vectors of neurons shown with a darker shading are moved closer to $\boldsymbol{x}(t)$ than neurons shown with a lighter shading.

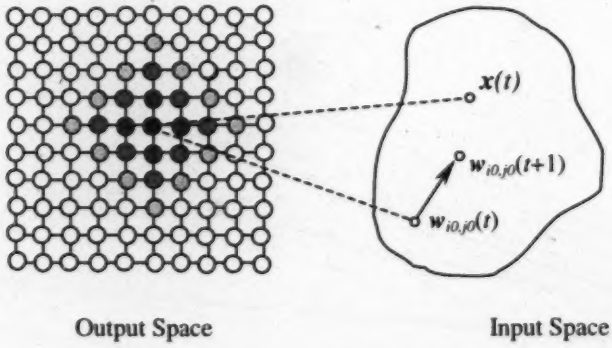


Figure 3.2 The training process in self organizing map

The complete training algorithm is shown below:

1. **Initialize network**
   For each node (i, j) set the initial weight vector $\mathbf{w}_{i,j}(0)$ to be random.
   Set the initial learning rate to 1, and set the initial neighborhood to cover 1/4 of the total neurons.

2. **Present input**
   Present $\mathbf{x}(t)$, the input pattern vector $\mathbf{x}$ at epoch t ($0 < t <$ T where T is the number of iterations defined by the user) to all nodes in the network simultaneously. $\mathbf{x}$ may be chosen at random or cyclically from the training data set.

3. **Calculate winning node**
   Calculate node (i_0, j_0) with smallest distance between weight vector and input vector

$$\left\|\mathbf{x}(t) - \mathbf{w}_{i_0, j_0}(t)\right\| = \min_{i,j}\left\|\mathbf{x}(t) - \mathbf{w}_{i,j}(t)\right\| \tag{3.1}$$

   thus:

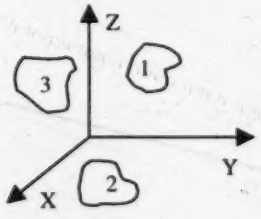
$$(i_0, j_0) = \arg\min_{(i,j)}\{\left\|\mathbf{x}(t) - \mathbf{w}_{i,j}(t)\right\|\} \tag{3.3}$$

4. **Update weights**
   Update weights for (i_0, j_0) and nodes within the neighborhood, using Formula 3.2.

$$\mathbf{w}_{i,j}(t+1) = \mathbf{w}_{i,j}(t) + \alpha(t) * N(i, j, i_0, j_0, t) * [\mathbf{x}(t) - \mathbf{w}_{i,j}(t)] \tag{3.2}$$

5. **Present next input**
   Decrease learning rate and neighborhood range. Then repeat step 2 choosing a new input vector $\mathbf{x}(t+1)$ until all iterations have been made ($t = T$).

Now let's see a very simple training example. The training data set is a 3-D synthetic data set with 3 clusters. Each cluster has 10 data points. The training data set is shown in Figure 3.3a. We use a 5 x 5 SOM. According to Figure 3.3b to Figure 3.3d, we can see how the SOM is trained to preserve the topology of training data. It is very clear that the input data are gradually organized to define areas on the map. For this simple data set and small SOM, it takes 10 iterations to organize the data points from the same cluster in the synthetic data set into a distinct cluster on the map. According to the Figure 3.3d, we can easily identify the existence of 3 clusters.



(a)

1,2	3	2	3	1,3
3		1,2		2
2	1,1	3	1	1
2	3	2	3	2,3
1	2,3	2	1	1,3

Training epoch = 0
(b)

1,1	1	1,2		2
1,1	2	1,2	1	2
1,3	3	2	2	2
1	3	3	3	2,3
3	3,3		3	2

Training epoch = 2
(c)

1,1	1,1	1	2,2	2,2
1	1,1	1	2	2,2
1			2,2	2
3	3			
3	3,3	3,3	3,3	3

Training epoch =10
(d)

Figure 3.3 The SOM training process for a simple synthetic data set
(a) The 3-D synthetic data set. The clusters are labeled with 1, 2 or 3.
(b) The map at the initial state.
(c) The map after 2 training iterations.
(d) The map after 10 training iterations

3.3 Parameter Influence

From the above discussion, we know that there are many parameters for SOM and its training. For any parameter, different values will have different effects on the final SOM. In this section, we discuss the most important parameters for SOM and SOM training.

3.3.1 Size of SOM

The size of the SOM determines the mapping "density". If the SOM size is too small, input data can only be mapped to a small number of neurons so that the preserved topology is very rough and only very distinct patterns can be distinguished by different neurons. On the other hand, if the SOM size is too large, the input data is mapped sparsely on the SOM. Some very similar patterns may be forced to map to different neurons. More importantly, the computational cost is proportional to the number of neurons in the SOM. A 20x20 SOM needs 4 times as much time to train as a 10x10 SOM. Thus we need to find a smaller SOM which is enough to preserve the topology of the original data. Usually, the size of SOM has to be determined by experiment, because we do not know the complexity of the topology in the input data before training. During experiment, we start from the maximum SOM size, which has approximately the same number of neurons as the number of input data. This ensures that the SOM will preserve the maximum complexity of topology for input data(Kohonen, 1995). In other words, if all the input patterns are distinct, there are still enough neurons to distinguish them. After we have trained a SOM with this maximum size, we can find the topology complexity of the input data. If the input data has distinct clusters, this means that the topology is not very complex, and we can try to reduce the size of SOM. After training with a smaller SOM, if the cluster boundaries become vague, this means that the topology of data can not be fully expressed by the smaller SOM, so we should use a larger SOM. In conclusion, if the computational cost is acceptable, the simplest and safest way is to always use the maximum size of SOM.

3.3.2 Learning Rate

The learning rate determines the learning and converging speed of SOM. If the learning rate is too high, there are oscillations during the learning process. If the learning rate is too low, the learning process is too slow and may not converge to the best point. When training begins, the connection weights are random numbers so the learning rate should be high. This will help capture the global topology of input data and skip the possible local minimum(Kohonen, 1995).

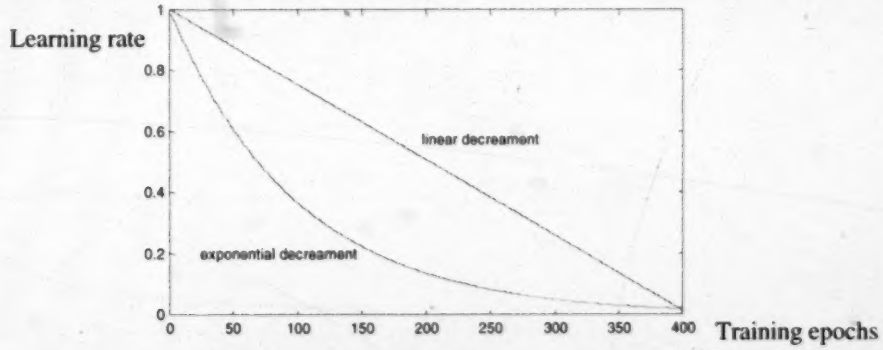


Figure 3.4 Two learning rate curves

As the global structure of SOM has come into being, the learning rate should be low so that the connection weights of SOM can be fine-tuned. In order to make the training converge smoothly, the learning rate should decrease during training. One simple technique is to decrease the learning

rate from 1 to a small number, say 0.01, linearly during training. The learning curve is shown in Figure 3.4. However, considering the fact that fine-tuning needs more iterations than rough learning, we may use a more sophisticated learning curve which is shown in Figure 3.4. In this curve, learning rate decreases exponentially during training. This learning curve assigns fewer learning epochs to rough learning(with high learning rate), and assigns more learning epochs to fine-tuning(with low learning rate).

3.3.3 Neighborhood Function

In SOM learning, each winning neuron has influence on its neighboring neurons. When the connection weight of the winning neuron is modified, its neighboring neurons should also get modified. The modification strength is modeled by the neighborhood function $N(i, j, i_0, j_0, t)$, where (i_0, j_0) denotes the coordinates of winning neuron, (i, j) denotes the coordinates of neighborhood neuron and t is the training epoch. This function should have the value of 1 in the center and decline gradually in the vicinity of the central neuron. The Gaussian function is usually used to satisfy this requirement, i.e.

$$N(i, j, i_0, j_0, t) = e^{-b*[(i-i0)^2+(j-j0)^2]}, \quad \text{where } (i - i_0)^2 + (j - j_0)^2 \le R(t)^2 \qquad (3.4)$$

In the above formula, the parameter b determines the attenuation rate in the neighborhood of the winning neuron. $R(t)$ is the neighborhood radius that declines over training, so the neighborhood range will decline during training. If b has a value close to 1, the winning neuron will have very small influence on its neighbors. If b has a value close to 0, the winning neuron will have significant influence on its neighbors. Figure 3.5 shows three profiles of the 2-D discrete Gaussian neighborhood function. In all three profiles, the radius of the neighborhood is 5. According to Figure 3.5, we can see how the neighborhood function affects the neighboring neurons. Generally, we choose $b = 0.05$, because it has a moderate decaying profile.

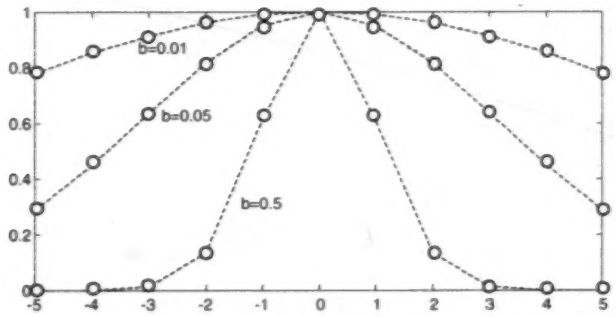


Figure 3.5 Three profiles of 2-D discrete Gaussian neighborhood functions
The winning neuron is in the center.

3.3.4 Neighborhood Size

Neighborhood size determines the range of influence of the winning neuron. Like the learning rate, neighborhood size should be large in the beginning and small in the fine-tuning phase. The minimum neighborhood size is 1, which refers to the immediate neighbors of the winning neuron. The starting neighborhood size should be large enough so that in the first few learning epochs, all the neurons get modified. This will ensure that the learning process captures the global topology of the input data. There is no strict requirement for the starting neighborhood size. Usually, the starting neighborhood size is set to 1/2 of the SOM size(Kohonen, 1995). For example, if the

SOM size is 20 x 20, the starting neighborhood size is generally set to 10 x 10. Like the learning rate, the neighbor size can also decrease linearly or exponentially over time(or epochs).

3.4 Interpretation of Results

SOM is represented by a 2-D array of neurons. Each neuron has a 1-D weight vector associated with it. So we can use a 3-D weight matrix to model the SOM data structure. During training, this 3-D weight matrix is modified and organized. After training, we can derive a collection of maps from this 3-D weight matrix.

3.4.1 Weight maps

The 3-D weight matrix has layers which corresponds to the training data dimension. This can be shown in Figure 3.6. Each layer has the same size as the SOM size. We can visualize these weight maps by displaying the weight matrix one layer at a time(Kaski, 1998).

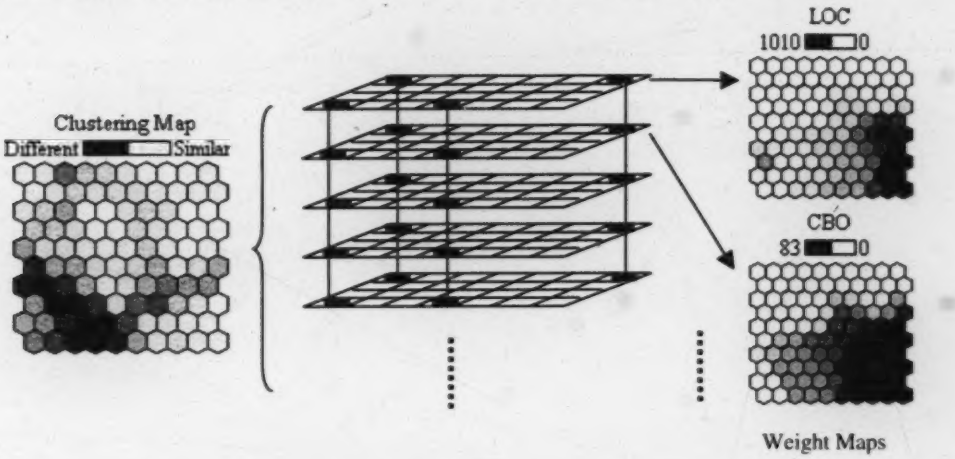


Figure 3.6 SOM weight matrix and its derived maps

This 3-D weight matrix has multiple layers corresponding to the multiple variables in the training data set. Each layer can be displayed as a weight map. The clustering map is calculated by combining all the layers. The training data set is a software measure data set from the Linguist software project, which is discussed in detail in section 4.2.2.

Each weight map represents one variable of training data. The most obvious information that we can get from these weight maps is association. If two weight maps are very similar, this means that the two variables they represent are highly interrelated. If two weight maps are very dissimilar, this means that the two variables they represent are not strongly interrelated. In addition, we can also determine the variable association of a data subset. For example, two weight maps are very similar in the upper-right corner but are very dissimilar in other area. This means that only the data located in the upper-right corner are highly interrelated.

3.4.2 Clustering map

From the weight matrix, we can derive another very important map which can be called the clustering map(Pedrycz, 2001a). This map shows the data clustering information of training data set. The following algorithm shows the steps to calculate the clustering map.

1. For each weight vector
   1.1 Compute the distances from immediate neighboring weight vectors to itself.
   1.2 Take the median of these distances and put a gray color corresponding to this median to the cell where this weight vector is located.
2. Cycle through all weight vectors.

The clustering map computed from the above algorithm represents the weight similarity distribution. If the weight vectors in one area are very similar, the color in this area is very bright which corresponds to low distance values(Kaski, 1999b). Clusters can be easily identified by looking for bright areas surrounded by dark boundaries. For some data set, there are distinct clusters, so in the clustering map, the dark boundaries are very sharp. However in some other data set, data are inherently scattered, so in the clustering map, we may not see clear dark boundaries.

3.4.3 Data distribution map

In the above discussion, we only use weight matrix to calculate the weight map and the clustering map. From this section, the data distribution map and the variable contribution must be calculated by combining weight matrix with the training data.

After the SOM is trained, we can map back all the training data to the trained SOM and record how many data points map to each neuron. Then we can get a data distribution map and know how data are distributed on the SOM. The algorithm to compute the data distribution map is listed below:

1. For each input data record
   1.1 Find the winning neuron corresponding to this input record.
   1.2 Increase the hit counts for this winning neuron.
2. Cycle to step 1 until all the input data have been checked.
3. Render the gray level color of each cell according to the hit counts of the neuron.

Combining the data distribution map with the clustering map, we can know how many data elements are in each cluster. Combining the data distribution map with the weight maps, we can get the histogram information(see detailed discussion in section 4.3.3).

3.4.4 Variable Contribution

The SOM is trained by multi-dimensional input data. Each variable of the input data will contribute to the learned SOM. However, different variables do not make the same contribution to the learned SOM. How significantly does a variable of input data contribute to the learned SOM? One quantitative way to show the contribution of a given variable is according to the following algorithm:

1. For each input data vector
   1.1 Find the winning neuron (i_0, j_0)
   1.2 Mask the given variable, and find the winning neuron (i_0', j_0') using the remaining variables
   1.3 Compute distance between the weight vectors of the two neurons (i_0, j_0) and (i_0', j_0')
2. Cycle through all input data vectors and compute the sum of the distances in step 4

The sum of the distances from step 5 shows how important a variable is for the learned SOM. If the distance sum is small, this means that omitting this variable does not make much difference, therefore this variable does not contribute very much to the learned SOM and can be called a redundant variable. If the distance sum is large, this means that omitting this variable will make much difference, therefore this variable contributes very much to the learned SOM and can be

called a significant variable. Using this indicator, we can identify the significant variables and the redundant variables, and do further analysis only on the significant variables.

3.5 Performance Measure

A self organizing map is a 2-D projection of highly-dimensional data. However, this projection is not unique. The same highly-dimensional data set can have many different SOMs. Each of these SOMs preserves the topology of the original data. For example, when we train a SOM with different parameters, we will get different shapes of maps. Even we use the same SOM parameters, because of the random initialization of weight matrix, we will still get different shapes of maps. How to compare different SOMs for the same training data set? Further, how to determine the association of two training data sets if we only know their SOMs? In order to answer these questions, some SOM performance measures are developed.

3.5.1 Accuracy

One goodness measure of SOM is how well the map can fit the learning data. It is the error between the trained SOM and the training data set. The SOM accuracy can be computed by the following steps:

1. For each record in input data set
   1.1 Find the winning neuron for this record.
   1.2 Compute the error(Euclidean distance) between this record and the weight vector of the winning neuron.
2. Cycle through all data vectors and sum all the errors to get the total error.

A more accurate SOM has a lower total error, and a less accurate SOM has a higher total error. This performance measure can be used as a termination criterion(see section 3.6.4 for details).

3.5.2 Robustness

After a SOM is trained, if we add some noise to the training data and map them back to the trained SOM, the winning neurons may change from the noise-free data. If a SOM is robust, it will not be sensitive to the noise. In order to compare the robustness of different SOMs, first we add some noise $\boldsymbol{e}$ to original data $\boldsymbol{x}$ and form the distorted data $\boldsymbol{x}'$. Then for each data record $\boldsymbol{x}_i$ and $\boldsymbol{x}_i'$, we find the winning neurons (i_0, j_0) and (i_0', j_0'), and compute the Euclidean distance D between the two neurons. The robustness can be calculated as the sum of D for all data records in $\boldsymbol{x}$ and $\boldsymbol{x}'$(Pedrycz, 2001a).

If a SOM is robust, it will not be influenced much by the noise, so the winning neurons of the distorted data is close to the winning neurons of the original data. If a SOM is not robust, it is noise sensitive, so the distorted data will have very different winning neurons from the original data. Using this performance measure, we can add some noise to the training data and observe the effects of SOMs with different parameters, so that we can tune the parameters to make SOM more robust.

3.6 SOM Improvement

The basic SOM learning algorithm is not complex, but in many cases, it can not give satisfactory results. There are many improvements that can be added to the basic SOM learning algorithm.

3.6.1 Data Normalization

Normalization is typically performed to control the variance of vector components or variables(Chakraborty, 2000). If some vector components have variance that is significantly higher than the variance of other components, those components will dominate the map

organization. In other words, some variables in the raw data may have larger scale and the values may be much larger than that for other variables. If this raw data set is inputted into SOM training, the weight vectors will be dominated by those variables with larger values and the final map will not reflect multi-dimensional topology. Instead, it will reflect the topology of those dominant variables. So the input data must be normalized before passing to SOM training. Usually normalization converts the variable values so that they no longer make any sense. The values are still ordered, but their range may have changed so radically that interpreting the numbers in the original context is very hard. Therefore, the normalization must be monotonic so that the saved information can be retrieved(de-normalization). Some frequently used normalization methods are: linear, logarithmic and logistic normalization.

3.6.1.1 Linear Normalization

This is the most frequently used normalization method. It is also the simplest one. With this normalization method, raw data are stretched to [0, 1]. Formula 3.5 shows how to do linear normalization.

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{3.5}$$

In the above formula, $\min(x)$ gives the minimum value for variable x, and $\max(x)$ gives the maximum value for variable x. We must do this normalization separately over all variables. This normalization approach works fine in many situations. However, sometimes when the training data set is skewed, this approach will have problems. For example, in the following data set, linear normalization makes things "worse".

Table 3.1 A synthetic data set

Observations	1	2	3	4	5	6	7	8	9
Variable 1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Variable 2	0.11	0.12	0.13	0.14	0.15	0.16	0.17	0.18	1
Variable 3	1	2	4	8	16	32	64	128	256

Table 3.2 The linearly normalized data for the synthetic data in Table 3.1

Observations	1	2	3	4	5	6	7	8	9
Variable 1	0	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1
Variable 2	0	0.011	0.022	0.034	0.045	0.056	0.067	0.079	1
Variable 3	0	0.004	0.01	0.03	0.06	0.12	0.25	0.5	1

In Table 3.1, the data in variable 1 is evenly distributed; the data in variable 2 is skewed to the low end; the data in variable 3 increase exponentially. According to Table 3.2, the linear normalization works well for the data in variable 1, but not works well in variable 2 and 3. The majority of data elements in variable 2 and 3 are compressed to close to zero by linear normalization, so that they will not have much influence on the SOM training. The training is dominated by variable 1 and the final topology will mainly determined by variable 1. In order to solve this problem, logarithmic normalization and logistic normalization are introduced.

3.6.1.2 Logarithmic Normalization

This normalization approach is suitable for exponentially increasing or decreasing data, because logarithm can cancel the exponential effect. In the following formula, it also adjust the normalized data to [0, 1].

$$x_{norm} = \frac{\log[x - \min(x) + 1]}{\log[\max(x) - \min(x) + 1]} \tag{3.6}$$

Table 3.3 shows the result of logarithmic normalization on the synthetic data in Table 3.1. It is obvious that logarithmic normalization works very well for variable 3. After logarithmic normalization, the exponential array becomes a linear array. However, the data in variable 2 is still not properly normalized. Most data elements in variable 2 are close to zero and will not contribute much to the SOM training.

Table 3.3 The logarithmically normalized data for the data in Table 3.1

Observations	1	2	3	4	5	6	7	8	9
Variable 1	0	0.16	0.31	0.45	0.57	0.69	0.8	0.9	1
Variable 2	0	0.016	0.03	0.05	0.06	0.077	0.09	0.11	1
Variable 3	0	0.125	0.25	0.375	0.5	0.625	0.75	0.875	1

3.6.1.3 Logistic Normalization

This normalization approach is particularly suitable for Gaussian distributed data. In the real world, most measurement data are approximately Gaussian distribution. This means that most data points located in the vicinity of the mean value, and there are few outliers far away from the mean value in the high end and/or the low end. For this type of data, it is suitable to apply neither linear normalization nor logarithmic normalization. Logistic normalization is designed to handle this situation. In Formula 3.7 and 3.8, first we normalize the input data by its mean and standard deviation, and get x_{scaled}. Then x_{scaled} is stretched to (0, 1), in such a way that it is approximately linear near the mean, but smoothly nonlinear at both ends.

$$x_{scaled} = \frac{x - mean}{std} \tag{3.7}$$

$$x_{norm} = \frac{1}{e^{-x_{scaled}}} \tag{3.8}$$

In the above formula, *mean* is the mean value over all the values of variable x. *std* is the standard deviation over all the values of variable x.

In Table 3.4, the logistic normalization is applied. We can see that data in variable 2 is normalized so that it could contribute to the SOM training significantly. Logistic normalization is frequently used in software measure data, because most software measure data sets are skewed to the low end.

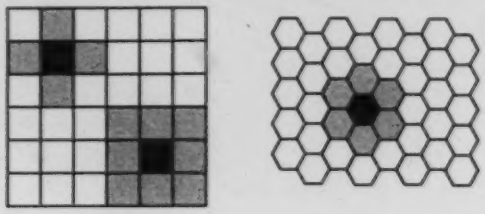
Table 3.4 The logistically normalized data for the data in Table 3.1

Observations	1	2	3	4	5	6	7	8	9
Variable 1	0.19	0.25	0.33	0.41	0.5	0.59	0.675	0.75	0.81
Variable 2	0.39	0.4	0.405	0.41	0.42	0.43	0.44	0.45	0.93
Variable 3	0.34	0.345	0.35	0.36	0.38	0.43	0.52	0.7	0.91

3.6.2 Neuron Visualization

In SOM visualization, there are basically two shapes of neuron(map cell): square and hexagon. The training principles are the same but hexagonal neurons are more reasonable to satisfy the SOM principle and can give a better visualization. In SOM training, every winner will affect its neighborhood neurons for a certain radius. In Figure 3.7 (a), for the square shape neurons, there

are two possible neighborhoods. In the upper left corner, all the immediate neighboring neurons have the same distance to the wining neuron, but the wining neuron is not completely surrounded by its neighboring neurons, so it will have an undesirable visual effect. In the lower left corner, the wining neuron is completely surrounded, but the immediate neighboring neurons do not have the same distance to the winner. On the other hand, in Figure 3.7 (b), the neurons are hexagonal. Each wining neuron is perfectly surrounded and all its neighboring neurons have the same distance to the winner. Therefore, the hexagonal neuron SOM can give us a better visualization.



(a) (b)

Figure 3.7 SOMs with different shapes of neurons

The basic SOM algorithm does not specify the shape of neuron, but the actual code to implement and visualize SOM is quite different for different neuron shapes. In order to implement the hexagonal neuron, much more programming effort is required. In section 7.5, we will see that both square shape and hexagonal shape neurons is implemented in the SOM Tool.

3.6.3 Frequency Sensitive

In the basic SOM learning algorithm, it is common to see that some winning neurons remain winning, while some other neurons remain idle. This will cause the weight maps to be non-continuous and introduce noise to the clustering map. In the data distribution map, we may see a large number of data records sitting in a single cell(neuron). This situation may be caused by the following factors:

- SOM size too large
  In SOM training, each input data record x will cause the winning neuron #k to be more similar to this data record. For another data record, the previous winning neuron #k may become the neighborhood of the new winning neuron #k', and then get modified. So in the next epoch, the same input data record x will have a chance to find a winning neuron #m which is different from #k in the previous epoch. However, when the SOM size is too large, the probability for a winning neuron #k to be modified as the neighborhood of another winning neuron #k' will become lower. So in the next epoch, the probability for a former winning neuron #k to become winner again will become higher. Thus the situation will occur that some winning neurons remain winning, while some other neurons remain idle.

- Neighborhood radius too small
  This is similar to the first factor. In the globally organizing phase, if the neighborhood radius is too small, the mutual influence of winning neurons is very weak. So for the same data record, it is more probable to find its former winning neuron than to find a different neuron.

- Inappropriate neighborhood function
  Neighborhood function determines the modification rates for neighboring neurons with different distance to the winning neuron. If the neighborhood function declines sharply in the

neighborhood of the winning neuron, the mutual interaction is weak, thus the same effect as in the case of a small radius above will occur.

To solve the problem, the frequency sensitive learning algorithm(Qiu, 1996) is applied. In this algorithm, each neuron has a counter that indicates the winning frequency of this neuron. When searching the winning neuron, this counter is a factor to be considered. If a neuron has a larger counter, it will have smaller probability to win. This frequency factor is reduced as learning epoch increases. For a small map, each neuron is easily affected either as a winner or as a winner's neighbor. So it is unnecessary to apply this algorithm in the case of a small size map. The following pseudo-code describes the frequency sensitive algorithm. In this pseudo-code, $w_{i,j}$ denotes the weight vector of the neuron with coordinates (i, j). Counter(i, j) denotes the frequency counter for the neuron with coordinates (i, j).

```
For each epoch
    For each input data record x
        Find winning neuron (i0, j0), which has minimum [Counter(i0, j0)*|| x - w_{i,j} || ]
        Increase the Counter(i0, j0) by a small number, say 0.01
        Modify neuron (i0, j0) and its neighborhood neurons
    End
End
```

3.6.4 Termination Criteria

SOM learning is an unsupervised learning. There is no way to compare the learning output to a target. So we must find a criterion to monitor the learning process so that we may know when to stop learning. In the basic SOM learning algorithm, after each learning epoch, the sum of weight changes is calculated. When the weights becomes stable, the sum of weight changes is very small. Then the learning process can be stopped. This termination criterion has some problems. Sometimes, the sum of weight changes may be very small, but the map has not learned sufficiently. During training, this problem can be found when the sum of weight changes goes to zero for some epochs and then goes up again. In order to deal with this problem, another termination criterion is used: matching error(or accuracy, see section 3.5.1 for details).

By monitoring this matching error during learning, we can see how the SOM fits the learning data set and how the error converges. For different learning parameters, the final matching error is different. Therefore, we can use this error not only as a termination criterion, but also use it to tune the SOM learning parameters.

3.6.5 Training Speedup

Training is the most time-consuming work in SOM application. For a small data set that has a few hundred records and a few variables, the time to train a SOM for this data set may be a few seconds. However, for a large data set with thousands of records and dozens of variables, the time to train a SOM might be a few hours. For huge database, which stores millions of records, the training time might be a few weeks. Therefore, developing an algorithm to speedup the training process is very helpful. Only a fast SOM training algorithm can make it possible to apply SOM in real business data analysis.

3.6.5.1 Neighborhood neuron indexing

This is a coding technique, but it can dramatically improve the training speed. During SOM training, once the winning neuron is selected, this neuron and its neighborhood neurons must be modified with different strengths. We know the index of the winning neuron and the radius of the

neighborhood, but we need to find out the indices of the neighborhood neurons and their associated adaptation strengths. A straight forward method is to calculate the indices of neighborhood neurons based on the relative location of the winning neuron. After determining the indices of neighboring, different strength coefficients are computed before modifying the neuron weights. We can see that heavy computational overheads are associated with this. These overheads are deeply nested inside the "For Loops" of the training algorithm, so the computational efficiency is greatly impacted. In order to solve this problem, we introduce the N-Table(Neighborhood Table). The N-Table stores all the neighborhood neuron indices and modification strengths. As shown in Figure 3.8, each row in the N-Table stores the neighborhood information for a particular neuron. This table is computed only when the neighborhood radius changes, which happens very infrequently during training. With this N-Table, the neighborhood neuron modification becomes trivial. First, according to the index of winning neuron, we directly go to the corresponding row in the N-Table, then we follow the arrow, and modify the neighborhood neurons using the supplied indices and strengths.

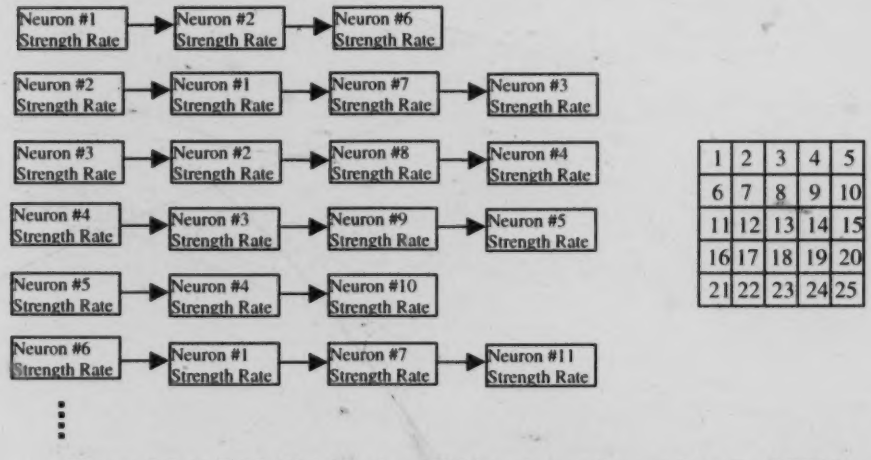

Neuron #1 Strength Rate
Neuron #2 Strength Rate
Neuron #6 Strength Rate
Neuron #2 Strength Rate
Neuron #1 Strength Rate
Neuron #7 Strength Rate
Neuron #3 Strength Rate
Neuron #3 Strength Rate
Neuron #2 Strength Rate
Neuron #8 Strength Rate
Neuron #4 Strength Rate
Neuron #4 Strength Rate
Neuron #3 Strength Rate
Neuron #9 Strength Rate
Neuron #5 Strength Rate
Neuron #5 Strength Rate
Neuron #4 Strength Rate
Neuron #10 Strength Rate
Neuron #6 Strength Rate
Neuron #1 Strength Rate
Neuron #7 Strength Rate
Neuron #11 Strength Rate
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25

Figure 3.8 N-Table(Neighborhood Table) when radius = 1 for a 5 x 5 SOM
In the left-hand side of the figure, the neurons on the map are labeled with a single number rather than a pair of coordinates

According to experiments, the N-Table can improve the training speed by a factor of 3~5 in both Matlab code and C++ code. The disadvantage of the N-Table is that it requires a lot of memory. For a SOM with square shape neurons, the required memory to store N-Table is listed in Table 3.5. We can see that for a small SOM(total number of neurons fewer than 20 x 20), the memory required to store the N-Table does not have any problem in an standard PC. When the SOM size is larger than 40 x 40, and when the neighborhood radius is large, the required memory is more than 1 megabytes, which could cause problems such as virtual memory swapping.

Table 3.5 Required memory to store N-Table for different SOM and different radius

Neighborhood Radius	Memory Required for N-Table(Byte)	
	20 x 20 SOM	40 x 40 SOM
1	15 K	60 K
2	40 K	160 K
4	130 K	520 K
8	460 K	1.84 M

3.6.5.2 Fast winner-searching by region

Most SOM training time is spent on the fine tuning phase, after the global structure of SOM has been formed. In this phase, we can accelerate the training by improving the searching for the winner which is the major training computation. When SOM is globally trained, we can divide the map into some regions. Each region is associated with a region weight vector that is equal to the weight vector of the central neuron in this region. When an input data record arrives, it is first compared to these region-weight-vectors. After we know which region wins, the next step is to compare the input data record to the weight vectors within the winning region. Because both the number of regions and the number of neurons within one region are small, the total number of comparison is small also. With this algorithm, the number of comparison is just the number of regions plus the number of neurons within one region. Otherwise the number of comparison is the total number of neurons(number of regions times the number of neurons within one region). In Figure 3.9, there is a 12x12 SOM with 144 neurons. In standard winner searching algorithm, the number of comparisons to find the winning neuron is 144. In fast winner searching algorithm, it first takes 16 comparisons to find the winning region, then it takes 9 comparisons to find the winning neuron. So the total comparisons in this algorithm is 25. It is obvious that this fast winner searching algorithm can greatly reduce the winner searching time.

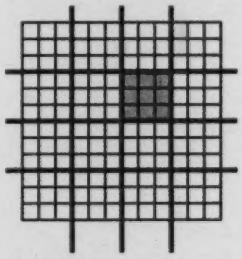


Figure 3.9 The SOM fast winner searching by region

3.6.5.3 Fast winner-searching by history

The above algorithm is based on the continuity of SOM, which means that neighboring neurons have similar weight vectors. There is another characteristic of SOM that we can use to accelerate the training. After SOM is globally organized, an input data record is very likely to find the same winning neuron in consecutive training epochs(Goppert, 1995). In other words, if an input data record finds that neuron #k is the winning neuron, then in the next epoch, this input data record will very likely find the same winning neuron(neuron #k). Using this characteristic, we can greatly reduce the SOM training time. Here is how it works:

1. First, the SOM is globally organized using basic training algorithm.
2. Then, for an input data record $\boldsymbol{x}_i$, it directly selects the winning neuron N_k that is the winning neuron for $\boldsymbol{x}_i$ in the previous epoch.
3. In order to confirm if N_k is really the winning neuron for $\boldsymbol{x}_i$ in this epoch, the neighborhood neurons of N_k are compared with $\boldsymbol{x}_i$, and a new winning neuron N'_k is found.
4. If N'_k is different from N_k, then winning neuron N_k is set as the new winning neuron N'_k, and step 3 is carried out next.
5. Winning neuron for $\boldsymbol{x}_i$ is confirmed as N_k.

In the above algorithm, we can see that the comparisons are made when the winning neuron is being confirmed. If a SOM has already been globally organized, this confirmation process will only take a few iterations. The essential advantage of this algorithm is that the number of comparisons does not depend on the size of SOM. This implies that even for a very large SOM,

say 100 x 100, we might still find a winning neuron using the same computation time as a much smaller SOM, say 20 x 20.

3.6.6 Using Different Distance Functions

The SOM algorithm is based on similarity comparison. In the basic SOM algorithm, similarity is defined as Euclidean distance between two vectors. If the input data record has the smallest Euclidean distance to a neuron weight vector, then this neuron is the winning neuron. However, similarity does not have to be defined as Euclidean distance. Similarity can be expressed by other distances as well. Formulas 3.9 to 3.11 are some frequently used distance functions.

$$\|\mathbf{a}-\mathbf{b}\| = \sqrt{\sum_{i=1}^{n}(\mathbf{a}_i - \mathbf{b}_i)^2} \qquad \text{(Euclidean Distance)} \qquad \textbf{(3.9)}$$

$$\|\mathbf{a}-\mathbf{b}\| = \max_i |\mathbf{a}_i - \mathbf{b}_i| \qquad \text{(Tchebychev Distance)} \qquad \textbf{(3.10)}$$

$$\|\mathbf{a}-\mathbf{b}\| = \sum_{i=1}^{n}|\mathbf{a}_i - \mathbf{b}_i| \qquad \text{(Hamming Distance)} \qquad \textbf{(3.11)}$$

In the above formulas, $\|\boldsymbol{a}-\boldsymbol{b}\|$ denotes the distance between $\boldsymbol{a}$ and $\boldsymbol{b}$. n denotes the number of variables in $\boldsymbol{a}$ and $\boldsymbol{b}$.

Different distance functions have different applications in expressing similarities. Two signals are similar if their Euclidean distance is small. Two filters are similar if the Tchebychev distance between their impulse responses is small. Two text strings are similar if their Hamming distance is small(Pedrycz, 2001b). Therefore, SOM does not necessarily use the Euclidean distance to compare the similarity between input data and the weights of neurons. We should apply different distance functions based on different type of data. If the input data are signals such as level of voltage, we apply Euclidean distance to SOM training algorithm. If the input data are text, such as newsgroup databases(Kohonen, 1999), we apply Hamming distance.

The following experiments show the effects of different distance functions for the Linguist software data set. The Linguist data set is discussed in detail in section 4.2.2. We apply three distance functions to three SOMs with the following parameters.

- Size of the maps: 10 x 10
- Initial condition: randomly initiated connections
- No frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 1000 iterations.

After training, we obtained three SOMs. The weight maps for the three SOMs are listed in Figure 3.10. We can see that using the three distance functions(Euclidean, Hamming and Tchebychev) in SOM generates similar effects(Similar distribution and association, see detailed discussion in section 4.3). However, if we test the robustness of SOM using these three distance functions, they make a difference. The robustness of SOM has been discussed in detail in section 3.5.2. We add different levels of Gaussian noise to the Linguist data set, then we compute the noise influence for SOMs using different distance functions. If the original data record is $\boldsymbol{x}$, then the distorted data record $\boldsymbol{x}'$ is $\boldsymbol{x}$ + k*N(0,1), where k is noise level and N(0,1) is an array of random numbers with Normal distribution. Table 3.6 shows the robustness of different SOMs. We can see that for different levels of noise, use of Euclidean distance and Hamming distance has a much lower noise

influence than Tchebychev distance, so use of them generates more robust SOMs than use of Tchebychev distance does.

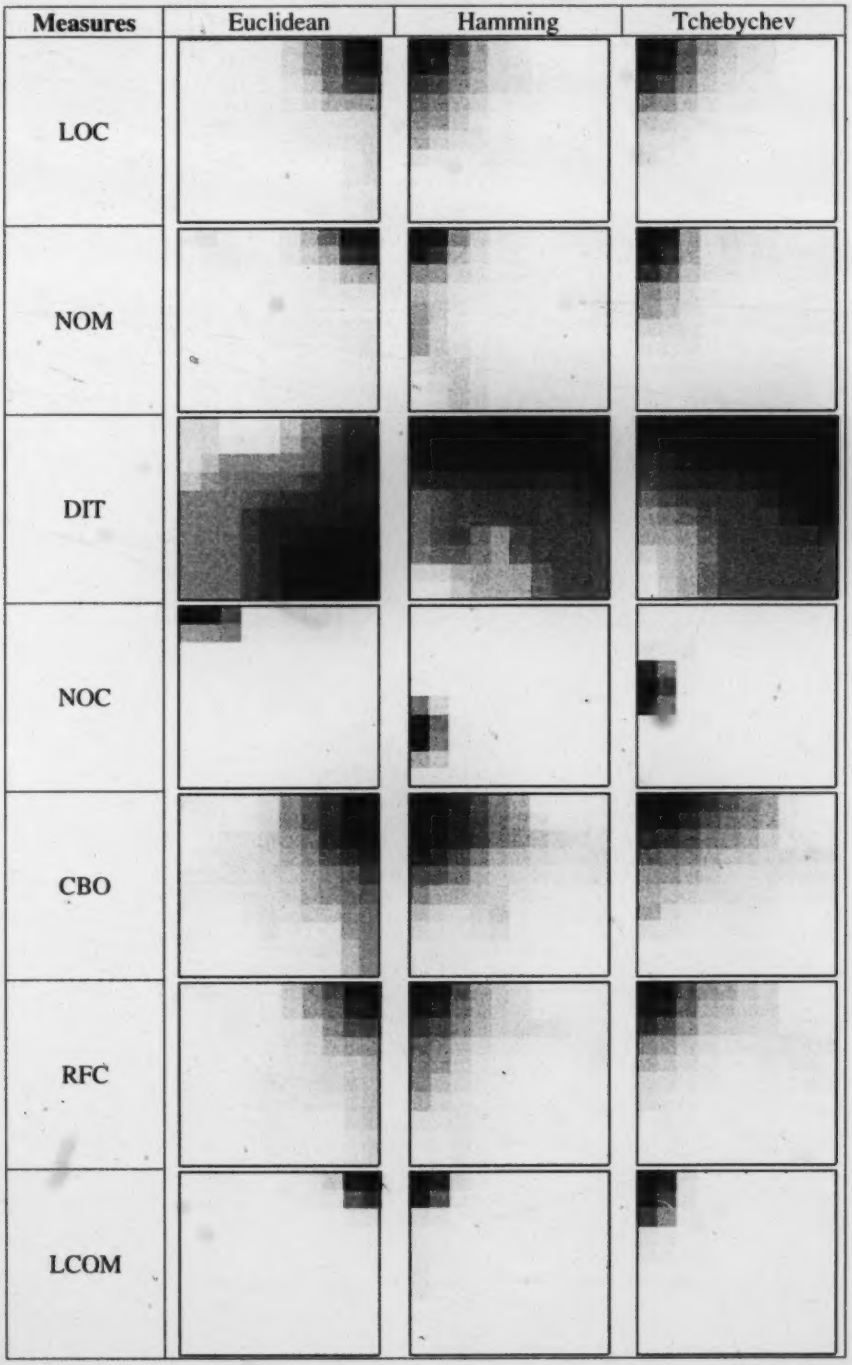


Figure 3.10 The SOM weight maps for the Linguist data set using different distance functions

Table 3.6 Noise influence of SOMs for the Linguist data set using different distance functions
Lower values indicate that the SOM is more robust to noise influence

Noise Level	0.05	0.1	0.15	0.2
Euclidean Distance	251.18	681.97	1033.61	1302.69
Tchebychev Distance	685.78	1069.22	1330.09	1549.26
Hamming Distance	329.38	685.38	975.24	1247.91

3.7 Combine SOM with Other Clustering Approach

From the above discussion, we know that SOM clustering is very suitable for completely unknown data. We can train a SOM to determine if there are distinct clusters, how many clusters there are and how big each cluster is. Although SOM is a quite powerful tool for data analysis, sometimes it can not completely satisfy our requirements. For example, once we know the clusters in a data set from SOM, we want to extract the prototype for each cluster. In SOM, there is no easy way to do this. We have to manually select all the neurons in a cluster and take an average of the neuron weight vectors to get a prototype. For a typical SOM with hundreds of neurons, this could be a very tedious work. We can however combine SOM with other clustering approaches to get prototypes. In section 2.8.3, we have discussed FCM as a clustering algorithm. Next, we discuss the combination of SOM with FCM to get cluster prototypes.

The idea of combining SOM with FCM is quite simple. Since in the FCM algorithm we must specify the number of clusters that we do not know yet, we can use SOM to find the number of clusters. So the steps to get cluster prototypes is listed below:

1. Train a SOM on the unknown data set
2. Identify number of clusters in the trained map
3. Train FCM on this data set using the number of clusters from step 2
4. Get prototypes from FCM and check them by mapping back to SOM

Now we use a synthetic data set to see how this combination of SOM with FCM works. The synthetic data set is a 3-D data set. It has four clusters each of which is generated by random numbers with Gaussian distribution. The whole data set is within the unit cubic, which is shown in Figure 3.11.

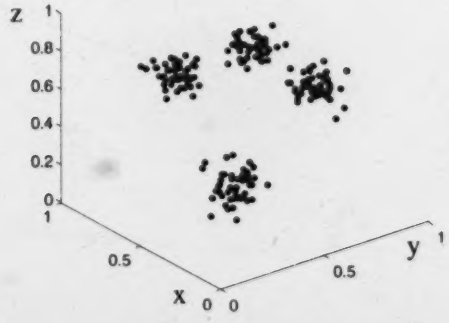


Figure 3.11 The 3-D synthetic data set, with 4 clusters, inside a unit cubic
Each cluster has 50 data points

First we use SOM to cluster this synthetic data set. We use a basic 10 x 10 SOM, following the general training setting discussed in section 3.3. The resulting clustering map is shown in Figure 3.12. It is very clear that there are 4 distinct clusters in the SOM clustering map. Then we specify 4 as the number of clusters and train the FCM to get 4 prototypes(cluster centers). The 4 cluster centers are mapped to the SOM as "+". We can see they are distributed in the 4 different clusters. If we do not use 4 as the FCM cluster number, FCM will not generate meaningful cluster centers(prototypes). The small triangles are the FCM cluster centers generated with cluster number equal to 3. The small squares are the FCM cluster centers generated with cluster number equal to 5. In both cases, there are cluster centers close to the cluster boundaries, which is meaningless.

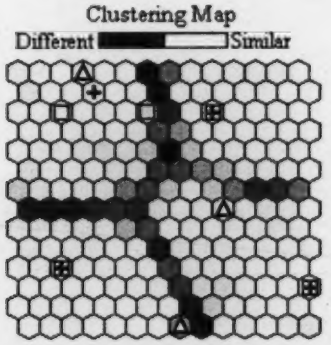


Figure 3.12 The SOM clustering map and the FCM cluster prototypes
The small "+" represent the prototypes computed from FCM with cluster number = 4. The small triangles and squares represent the prototypes computed from FCM with cluster number = 3 and 5, respectively.

According to the above experiment, SOM gives the correct number of clusters and FCM gives the precise prototypes. Without SOM, we might use the wrong number of clusters and generate wrong prototypes. Without FCM, we have to do a lot of manual work and might not find the best prototypes.

4 Application of SOM to Software Measure Analysis

In this chapter, we discuss the application of SOM in software measure analysis. First, a comparison between SOM and statistical analysis is discussed. Then we use some real software measure data to demonstrate the uniqueness and effectiveness of SOM analysis in software measures.

4.1 Comparison with Conventional Analysis

The conventional analysis of software measures is by statistical means, which provides a large number of indicators to describe the data. Some of most frequently used statistical indicators include mean, median and variance. With SOM, we use a completely different way to describe software measure data. Some of the most frequently used SOM analysis methods include the weight maps, the clustering maps and the distribution maps. Next, we discuss the advantages and disadvantages of each analysis approach.

- Distribution Analysis
  When we analyze software measure data, the first step is to determine the data distribution. In other words, we want to know if the data is evenly distributed over the full range, or skewed to one end.

  In conventional analysis, we use minimum, maximum, mean and variance to show the distribution characteristics of data. In addition to the above numerical indicators, we also use box plots and histograms to visualize the data distribution information. On the other hand, in the case of SOM, we combine the data distribution maps with the weight maps, then we can determine the data distribution information directly from these maps. This is discussed in detail in section 4.3.3.

  Conventional analysis generates numbers(mean, variance, etc.), which are not easy to interpret. It also generates graphs(such as box plots and histograms), which are easy to understand. However, these graphs are single-purpose, which means that they are only used to show the data distribution. In case of SOM, we can easily determine the data distribution information by combining the data distribution maps and the weight maps, and these maps are not single-purpose. For example, by combining the data distribution maps with the clustering maps, we get the cluster distribution information. By combining the weight maps with the clustering maps, we get the cluster profile information. By comparing the weight maps, we get the data association information.

- Association Analysis
  Association analysis is concerned with how closely different data variables are interrelated. In software measure analysis, we want to find out if there is strong associations between two software measures in a particular software project. For example, in the MIS(Munson 1996) data set, we found that LOC(Lines of Code) is strongly interrelated with TChar(Total Characters). This strong association tells us that in this MIS project, we can use LOC to indicate TChar, or vice versa. It also implies that there is information redundancy in this data set. If necessary, we can remove the redundant software measures.

  Conventional analysis employs the correlation matrix to represent the associations between any two software measures. The correlation matrix is a 2-D matrix of numbers that are between 0 to 1. The rows and columns correspond to different measures in the software measure data set. For example, if we want to find the association strength between measure #2 and measure #4, then we look at the correlation matrix and find the intersection of row #2

and column #4. If the number in the intersection is close to 1, this means that the two measures are closely interrelated. If the number in the intersection is close to 0, this means that the two measures are not interrelated at all. If the number in the intersection is negative, this means that the two measures are inversely interrelated(Anderson, 1984).

By means of SOM, as we have discussed in section 3.4.1, we can determine the association between any two measures by comparing the similarity of the two weight maps corresponding to the two software measures. The similarity refers to the image similarity. Identifying association by comparing similarity is quite easy and straightforward for humans, while reading numbers from correlation matrix can be an annoying task.

- Cluster Analysis
  In conventional analysis of software measures, clustering is not an easy task. There are some advanced clustering approaches, such as FCM(Bezdek, 1981), but they require some assumption about the data, such as number of clusters. SOM is inherently suitable for complex data cluster analysis. From the clustering map of SOM, we can easily know if there are distinct clusters and identify how many clusters there are in the data set. More importantly, by combining the data distribution map and the weight maps, we can easily determine the characteristics of each cluster(see detailed discussion in section 4.4.

- Noise Resistance
  Because of the uncertain nature of software objects, there are a lot of "noises" (outliers) in the raw software measure data set. Outliers are very common in any software measure data set, and they will seriously affect the data analysis, such as distribution and association analysis. In conventional analysis, we try to identify and eliminate outliers, usually based on probability. SOM, because of its nonlinear neural algorithm, can handle outliers smoothly in a way similar to the way that the human brain processes noises.

- Accuracy
  Conventional analysis of software measures deals with numbers, while SOM analysis deals with maps(like images). One might say that numbers are the most accurate indicators to describe objects. For some aspects of objects, such as weight and length, numbers are obviously more accurate than maps. However, for software, we usually want to know such aspects as complexity and quality. These aspects of software are inherently not well defined, so using exact numbers does not make much sense. Using SOM, we can obtain an overall image of the software through a collection of maps. Although it is not as accurate as statistical numbers, considering the characteristics of software measures, SOM is very suitable for software measure analysis.

- Expressiveness
  As we have discussed in the above sections, conventional analysis of software measure data needs a large number of numerical indicators and graphical plots to get an overall description of the data. In case of SOM, all we generate is a 3-D weight matrix, from which we can derive three basic types of maps: the clustering maps, distribution maps and the weight maps. Each of these maps contains a large amount of information. By combining these maps, we can get a comprehensive understanding of the software measure data. In our daily life, it is very obvious that maps or images are more expressive than numbers or words. For example, a person can be easily identified by looking at his or her face photo. On the other hand, you might never identify that person by reading the text description of his or her face.

4.2 *Description of the Experimental Data*

We use some experiments to demonstrate the application of SOM in software measure analysis. There are three software projects involved: MIS, Linguist and JDK. The MIS data set was collected during the development and maintenance of a Medical Imaging System(Munson, 1996). The Linguist and JDK data were extracted from the source code of the two projects by WebMetrics(Succi, 1999). WebMetrics is a software measure collection tool. It is a web based system and can extract software measures from multiple source languages: C/C++, Java, Smalltalk, Rose Petal and COBOL. The simplified process of collecting software measure data by WebMetrics is shown in Figure 4.1.

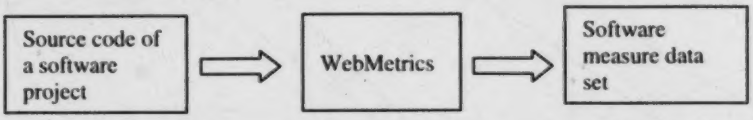


Figure 4.1 The simplified process of collecting software measures data by WebMetrics

The MIS data set is used in data distribution and association analysis. The Linguist data set is used in single SOM cluster analysis. The JDK data set is used in hierarchical SOM cluster analysis.

Table 4.1 Software measures used in the MIS data(Munson, 1999)

Measure	Description
LOC	Number of lines of code, including comments
CL	Number of lines of code, excluding comments
TChar	Number of characters
TComm	Number of comments
MChar	Number of comment characters
DChar	Number of code characters
N	Program length $N = N_1 + N_2$, N_1 is the total number of operators, N_2 is the total number of operands
N'	Estimated program length $N' = n_1 \log_2 n_2 + n_2 \log_2 n_2$, n_1 is the number of unique operators, n_2 is the number of unique operands
N_F	Jensen's estimator of program length $(\log_2 n_1)! + (\log_2 n_2)!$, n_1 and n_2 are the same as above
V(G)	McCabe's cyclomatic number is one more than the number of decision nodes in the control flow graph
BW	Belady's bandwidth measure $BW = 1/n \Sigma_i (i * L_i)$, L_i represents the number of nodes at level i in a nested control flow graph of n nodes. This measure is indicative of the average level of nesting or width of the control flow graph representation of the program

4.2.1 MIS Data Set

The MIS is a commercial software system consisting of approximately 4500 routines written in about 400,000 lines of Pascal, FORTRAN, and PL/M assembly code. The MIS development took five years, and the system has been in commercial use at several hundred sites for quite a few years. The MIS data were collected as the number of changes made to each module due to faults discovered during system testing and maintenance, along with 11 software complexity measures for each module that comprises MIS. In this thesis, MIS is represented by a subset of the whole MIS data with 390 modules written in Pascal and FORTRAN. These modules consist of approximately 40,000 lines of code. The 11 software measures are listed in Table 4.1. In this data

set, each module has a label, "Changes", which is the number of modifications made to this module during development and maintenance.

4.2.2 Linguist Data Set

The Linguist is a framework and organizational tool for Java. It provides a set of structured classes that assist in the job of organizing applets, servlets or applications of any size. This project was written in Java. It has 643 classes. Table 4.2 shows the software measures used to measure the classes in the Linguist project. All the software measures listed below have been introduced in section 2.5. We can see that in the Linguist data, all the software measures except LOC are OO(Object Oriented) software measures. In the previous section, the MIS project was written in Pascal and FORTRAN, which are structured languages. So it is quite natural to use non-OO software measures for the MIS data collection. the Linguist was written in Java, which is a pure OO language. So we must apply OO software measures for the Linguist data collection. Although LOC is not a pure OO software measure, it is the most frequently used, generic software measure, so we still use it for this OO project.

Table 4.2 Software measures used in the Linguist and the JDK data

Measure	Description
LOC	Lines of Code
NOM	Number of Methods
DIT	Depth of Inheritance Tree
NOC	Number Of Children
CBO	Coupling Between Objects
RFC	Response For Class
LCOM	Lack of Cohesion Of Methods

Table 4.3 Major source code packages provided by JDK

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.beans	Contains classes related to Java Beans development.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.math	Provides classes for performing arbitrary-precision integer arithmetic and arbitrary-precision decimal arithmetic.
java.net	Provides the classes for implementing networking applications.
java.rmi	Provides the RMI package.
java.security	Provides the classes and interfaces for the security framework.
java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes.
javax.swing	Provides a set of "lightweight" components.
org.omg.CORBA	Provides the mapping of the OMG CORBA APIs to the Java programming language.

4.2.3 JDK Data Set

The JDK data set is from the Java 2 SDK Standard Edition, published by Sun Microsystems, Inc. The Java 2 installation archive file "src.jar" includes Java programming language source code for all classes that make up the Java 2 core API. This source code is provided for training purposes, to help developers learn and use the Java programming language. These classes do not include platform-specific implementation code. The major packages provided in this API library are listed in Table 4.3.

There are 3016 classes in this data set. We use the seven software measures as listed in Table 4.2 to describe each class and this JDK data set to do hierarchical SOM clustering. That is, we generate a small map using the whole data set, then select an interesting cluster and grow a new SOM based on the data in this cluster. Repeating this process, we can build a hierarchical SOM.

4.3 Distribution and Association Analysis

From a trained SOM, we can derive a collection of maps, from which we can directly determine the data distribution and association information. Because software is a complex entity and difficult to describe by numbers, this graphical analysis is particularly suitable for software measure analysis.

4.3.1 Statistics

Before we start to use SOM to analyze this MIS data, we show some statistical indicators for later comparisons to the use of SOM method. We show the range, mean, median, standard deviation and histogram for each software measure. We also show the correlation matrix for all software measures.

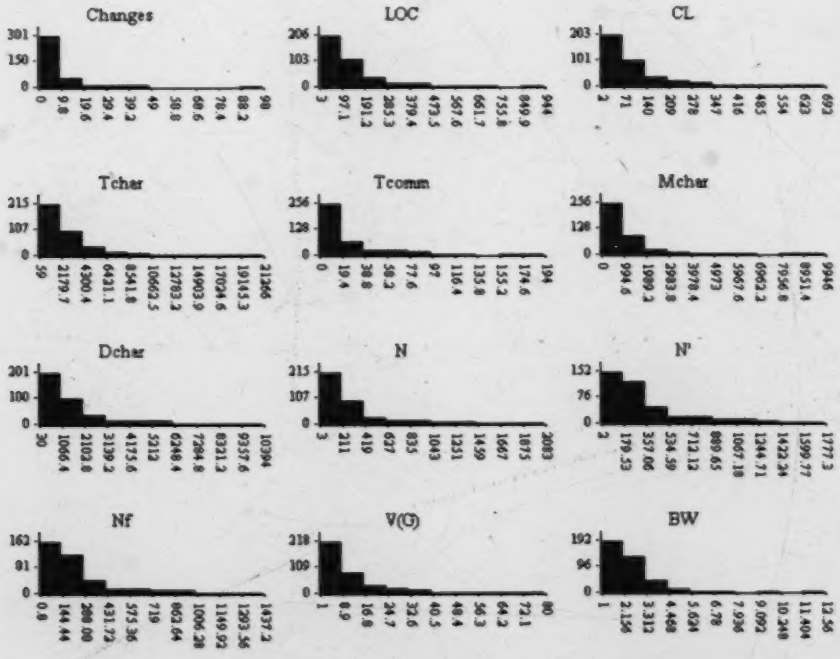


Figure 4.2 Histograms for all the software measures in the MIS data

Table 4.4 Statistics for the MIS data

Measures	Minimum	Maximum	Mean	Standard Deviation
Changes	0	98	7.25	10.42
LOC	3	944	130.86	134.07
CL	2	692	103.95	110.51
Tchar	59	21266	2819.7	2924.4
Tcomm	0	194	23.96	27.37
Mchar	0	9946	974.81	1178.6
Dchar	30	10394	1647.3	1808.5
N	3	2083	298.27	331.98
N'	2	1777.3	342.66	321.86
N_f	0.8	1437.2	260.02	258.49
V(G)	1	80	71.91	12.97
BW	1	12.56	2.44	1.41

Table 4.5 Correlation matrix for MIS data

	1	2	3	4	5	6	7	8	9	10	11	12
1 Changes	1.00	0.73	0.68	0.72	0.75	0.66	0.69	0.62	0.66	0.66	0.68	0.26
2 LOC	0.73	1.00	0.99	0.96	0.85	0.82	0.94	0.91	0.90	0.90	0.85	0.41
3 CL	0.68	0.99	1.00	0.93	0.80	0.74	0.94	0.92	0.90	0.90	0.84	0.42
4 Tchar	0.72	0.96	0.93	1.00	0.86	0.90	0.94	0.92	0.90	0.91	0.85	0.42
5 Tcomm	0.75	0.85	0.80	0.86	1.00	0.80	0.82	0.80	0.81	0.81	0.82	0.33
6 Mchar	0.66	0.82	0.74	0.90	0.80	1.00	0.71	0.68	0.67	0.67	0.67	0.30
7 Dchar	0.69	0.94	0.94	0.94	0.82	0.71	1.00	0.97	0.96	0.96	0.87	0.45
8 N	0.62	0.91	0.92	0.92	0.80	0.68	0.97	1.00	0.95	0.95	0.86	0.44
9 N'	0.66	0.90	0.90	0.90	0.81	0.67	0.96	0.95	1.00	1.00	0.85	0.44
10 Nf	0.66	0.90	0.90	0.91	0.81	0.67	0.96	0.95	1.00	1.00	0.85	0.44
11 V(G)	0.68	0.85	0.84	0.85	0.82	0.67	0.87	0.86	0.85	0.85	1.00	0.61
12 BW	0.26	0.41	0.42	0.42	0.33	0.30	0.45	0.44	0.44	0.44	0.61	1.00

4.3.2 SOM Training

The SOM size is set to 20 x 20. In section 3.3.1, we have discussed how to determine the size of SOM. For an unknow data set, the safest way to determine the SOM size is to set the total number of neurons to be close to the total number of training data records. In the MIS data set, there are 390 data records, so we choose a 20 x 20 SOM, which is a square map and has the number of neurons closest to 390. The initial neighborhood radius is usually set to 1/2 of the SOM radius(Kohonen, 1995), so in this experiment we set the initial radius as 5. Random initialization is used so that the weight matrix of SOM is initialized to random numbers between 0 to 1. We use logistic normalization so that the normalized data will have flatter histograms. The learning rate is set to 1 in the begining and allowed to decline to 0.2 in the global organizing phase, then in the fine tuning phase, the learning rate will decline from 0.2 to 0.02. There are 500 training epochs for the rough training phase, and 2000 epochs in the fine tuning phase. The distance function used in the SOM algorithm is Euclidean distance. The neighborhood decline rate is set to 0.05(See discussion in section 3.3.3 for details).

After training, we obtained some maps which can be used to do various data analysis. In this experiment, we focus on the software measure data distribution and association analysis, so the data distribution map and the weight maps are shown as in Figure 4.3 and Figure 4.4.

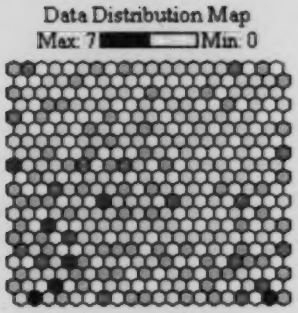


Figure 4.3 The data distribution map for the MIS data

4.3.3 Distribution

Data distribution information can be obtained by combining the data distribution map in Figure 4.3 and the weight maps in Figure 4.4. First let's observe the data distribution map. We can see that data are scattered all over the map. In the lower half of the map, there are slightly more data points than in the upper half of the map. Then let's move to the weight maps. The number of the weight maps corresponds to the number of variables(measures) that we are analyzing. The dark colors represent high values, and the light colors represent low values. Data distribution analysis must be done independently for each variable(measure). We use LOC as an example to explain how to determine data distribution information from these maps. In the weight map for LOC, most dark colors are in the upper-right corner. Nearly 3/4 area of the map is occupied by a narrow range of light colors. Together with the data distribution map, where only a small part of data are located in the upper-right corner, we can conclude that most data points(software modules) have low values(light colors), and lie within a narrow range. This analysis result can be easily supported by the histogram in Figure 4.2. For other variables(software measures), we can follow the same process to analyze their distribution.

Globally speaking, all the weight maps have much more light color area than dark color area. This means that in this MIS data set, the number of small modules is far more than the number of large modules.

4.3.4 Association

Data association information can be obtained from the weight maps. This is a rather simple process. If we want to know the association between two software measures, we just compare the two weight maps corresponding to these two measures. If the two maps are very similar, then the two software measures are closely interrelated. If the two maps are very different, then the two software measures are not strongly interrelated. Comparing the similarities between two maps(images) is quite easy for humans, while it may require sophisticated mathematical computation for computers. For example, weight maps for LOC and CL are very similar(almost identical). This means that the two software measures LOC and CL are closely interrelated. When we examine correlation matrix in Table 4.5, the correlation between LOC and CL is 0.99.

For the weight map of Changes, we cannot find another map which is very similar to it, but we can still find that the weight map for TComm is more similar to the weight map of Changes than other maps. If we examine the correlation matrix in Table 4.5, we can find that the correlation between TComm and Changes is 0.75, slightly higher than the correlation between any other software measures.

The weight map for BW is very different from other maps. It is the only map that has dark colors in the lower-right corner. This indicates that BW is a unique software measure. It reveals some unique aspects of software modules that other measures do not.

Globally speaking, there is a lot of redundant information in this MIS data set. Many software measures are closely interrelated. Large modules tend to have more Changes, but this may not be the case for a small number of "outlier" modules.

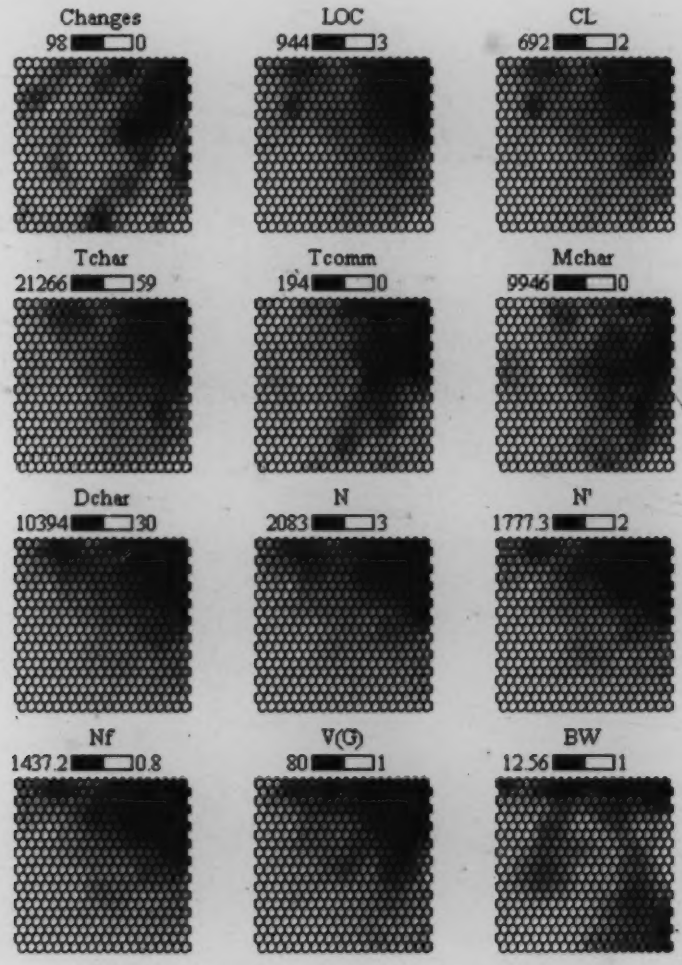


Figure 4.4 The weight maps for the MIS data

4.3.5 Summary

From the above experiment, we can see the unique way in which SOM can be used to do data distribution and association analysis. Using SOM's data distribution map and weight maps, we can determine the data distribution and association information simultaneously. In conventional software measure analysis, distribution and association analysis is done separately by two completely different approaches: histograms and correlation matrix. Another obvious advantage of SOM association analysis is that it expresses associations by means of map(image) similarities,

which is very direct and intuitive. Using conventional association analysis, we have to locate and read numbers from correlation matrix, which in many cases is undesirable.

4.4 Cluster Analysis

As we have discussed in section 2.8.3, clustering plays an important role in software measure analysis. There is a limited, but still significant, body of knowledge of relations between internal software measures and external properties of the resulting product(Pedrycz, 2001a). Therefore, if we find some software modules are in the same cluster in terms of their internal software measures, then we can predict that they are very likely to exhibit similar external behaviors. More importantly, if we have known the behavior of some modules in a cluster, then we can predict the behavior of other modules in the same cluster. Clustering all the software modules in a software project will greatly benefit us in managing and maintaining these software modules. In the following sections, we first discuss the application of single SOM clustering, then we find out how to organize a large data set into a hierarchical SOM.

4.4.1 Clustering by Single SOM

We use the Linguist data set in this section. First, we need to set the parameters for SOM including training. Because the Linguist data set has 643 data records, we use a 25 x 25 SOM(see detailed discussion about size selection in section 3.3.1). Other parameters are listed below:

- Normalization of variables: logistic normalization
- Initial neighborhood radius: 6
- Initial condition: randomly initiated connections
- Frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 3000 iterations

After training, the weight maps and the clustering map are stable. Figure 4.5 shows the SOM clustering map for the Linguist data set. Figure 4.6 shows the SOM data distribution map for the Linguist data set. Figure 4.7 shows the SOM weight maps for all software measures in the Linguist data set.

We can see obvious dark boundaries in the clustering map. These boundaries distinguish the clusters. There are approximately 5 clusters in the clustering map, as shown in Figure 4.5. In order to know how many data records are in each cluster, we go to the data distribution map in Figure 4.6. We can see that data records are distributed evenly except in the case of a few "hot" neurons, where many data records gather. These "hot" neurons are highly homogeneous clusters, and are listed in Table 4.6. We can see that the software modules(Java classes) corresponding to a "hot" neuron all have identical software measures. This suggests that these modules must have strong relations in their external behaviors. Observing their descriptions, we can support this relation. For example, all the modules in neuron "B" are dealing with exceptions.

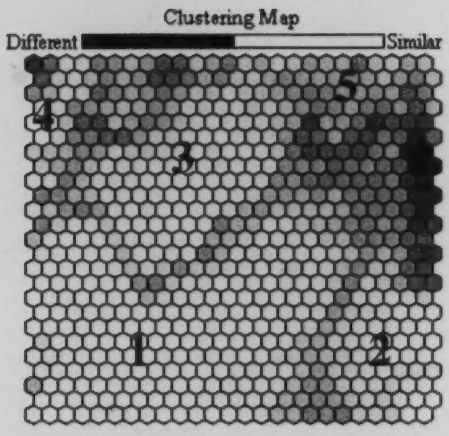


Figure 4.5 The SOM clustering map for the Linguist data
There are five distinct clusters, as labeled on the map

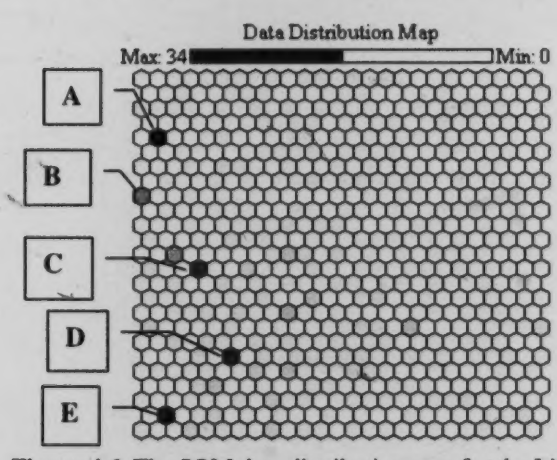


Figure 4.6 The SOM data distribution map for the Linguist data
There are some "hot" neurons(highly homogeneous clusters) which are labeled for later analysis

According to the data distribution map, we know that cluster #1 contains the largest population of data records, and cluster #5 contains the smallest population of data records. Cluster #1 has more "hot" neurons than any other cluster.

We now examine what kind of data are located in each cluster. We combine the data clustering map with the weight map for each software measure. Then we can obtain the profile for each cluster. For example, Cluster #2 is in the lower-right corner, where there are just light colors in all 7 weight maps. This means that in Cluster #2, all software modules have low values in terms of the 7 software measures. We can apply the same analysis steps as in the case of Cluster #2 to other clusters.

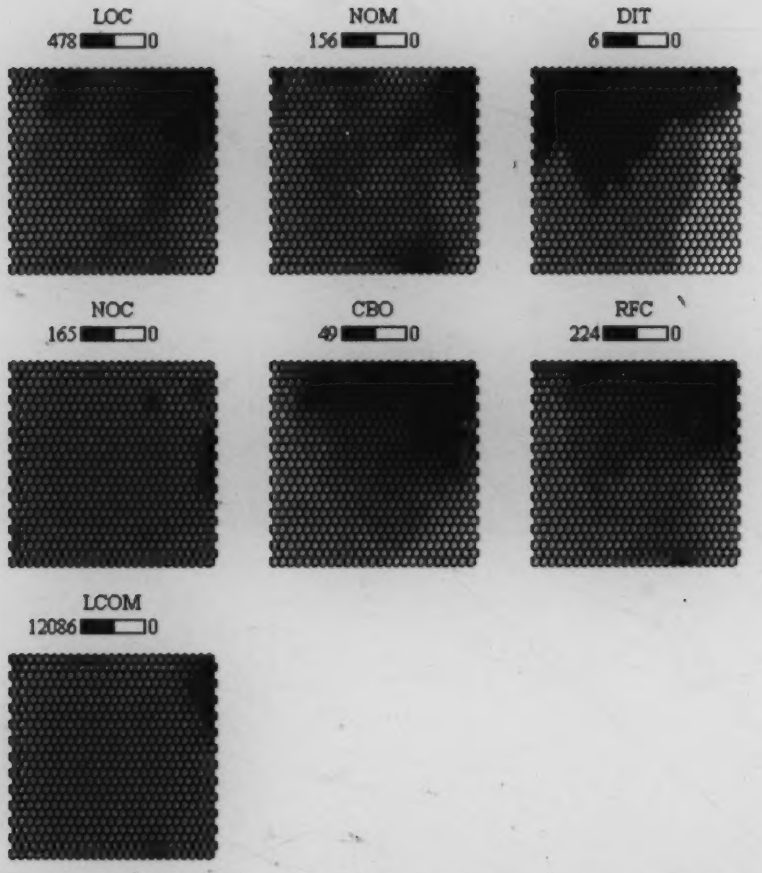


Figure 4.7 The SOM weight maps for all software measures in the Linguist data

Another question regarding the cluster analysis is to find out the software measures that influence the shapes of the clusters. In other words, we want to find out the software measures that contribute most significantly to forming the clustering map. If we compare the clustering map in Figure 4.5 with the weight maps in Figure 4.7, we readily observe that the clusters in the clustering map match very well in the color regions of the weight map of DIT. Cluster #1, #2, #3 and #4 correspond to different colors(DIT values) in the DIT weight map perfectly. Cluster #5 does not correspond to the colors in the DIT weight map. This implies that Cluster #5 is not derived from the DIT weight map, but from a combination of other weight maps.

Why does DIT have so much influence on the clustering map? If we carefully go through the weight maps, we can see that the DIT weight map is quite different from other weight maps. In other weight maps, most neurons are light colored, which means that most software modules have low values in these software measures. In the DIT weight map, different colors are obviously distinguishable. None of the colors in the DIT weight map occupies too much area. This means that all software modules are distributed relatively evenly over different DIT values. This can be easily seen from the histogram in Figure 4.8

Table 4.6 The "hot" neurons(highly homogeneous clusters) in Figure 4.6

Neuron	Number of software classes	Common description in software class name
A	34	All the classes have the same software measure values: LOC=5, NOM=5, DIT=4, NOC=0, CBO=2, RFC=7, LCOM=10 Common name description is ".keyword.", which means that these classes are in the keyword package.
B	14	All the classes have the same software measure values: LOC=1, NOM=1, DIT=4, NOC=0, CBO=1, RFC=2, LCOM=0 Common name description is "Exception", which means that these classes are dealing with exceptions.
C	20	All the classes have the same software measure values: LOC=5, NOM=2, DIT=2, NOC=0, CBO=1, RFC=3, LCOM=1 Common name description is ".handler.", which means that these classes are in the handler package.
D	22	All the classes have the same software measure values: LOC=4, NOM=3, DIT=2, NOC=0, CBO=2, RFC=5, LCOM=3 Common name description is ".value.", which means that these classes are in the value package.
E	25	All the classes have the same software measure values: LOC=7, NOM=2, DIT=2, NOC=0, CBO=2, RFC=3, LCOM=1 Common name description is ".handler.", which means that these classes are in the handler package.

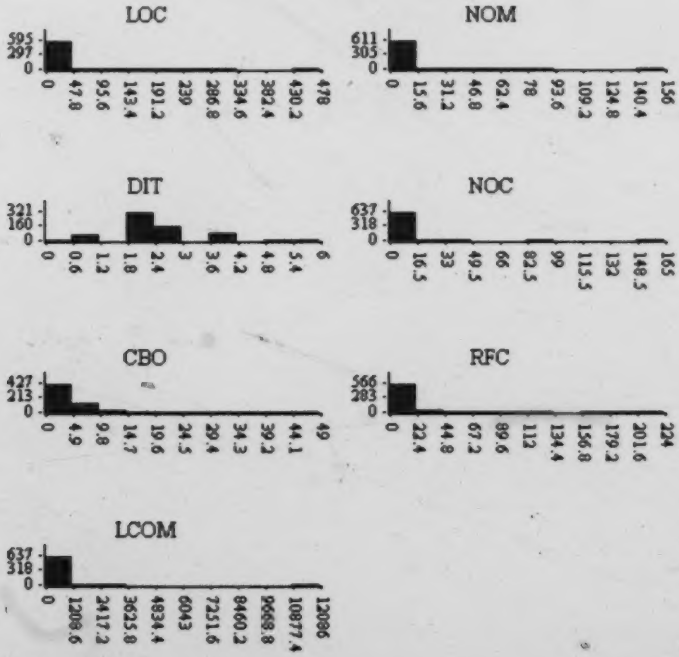


Figure 4.8 Histograms for all software measures in the Linguist data
This figure shows that all measures except DIT are skewed to the low end

To summarize, DIT has the biggest influence in clustering the software modules, because it is the only measure that can differentiate clusters in the otherwise homogeneous area. Then what happens if we remove DIT from the Linguist data and use the remaining measures to do clustering?

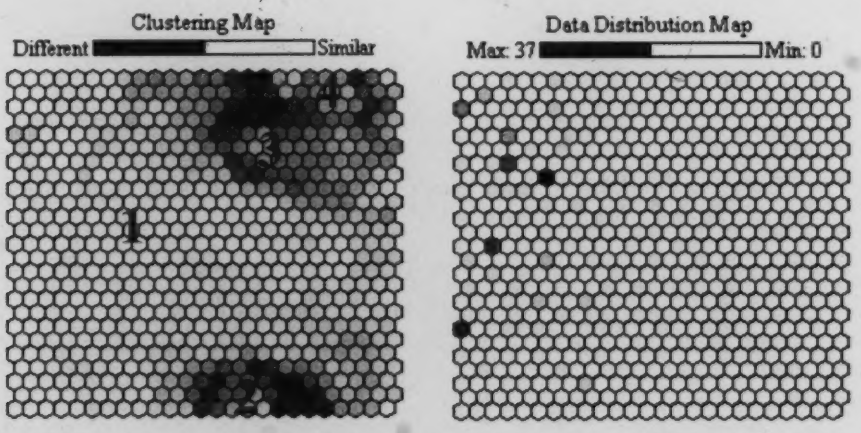


Figure 4.9 The clustering map and the data distribution map for the Linguist data without DIT

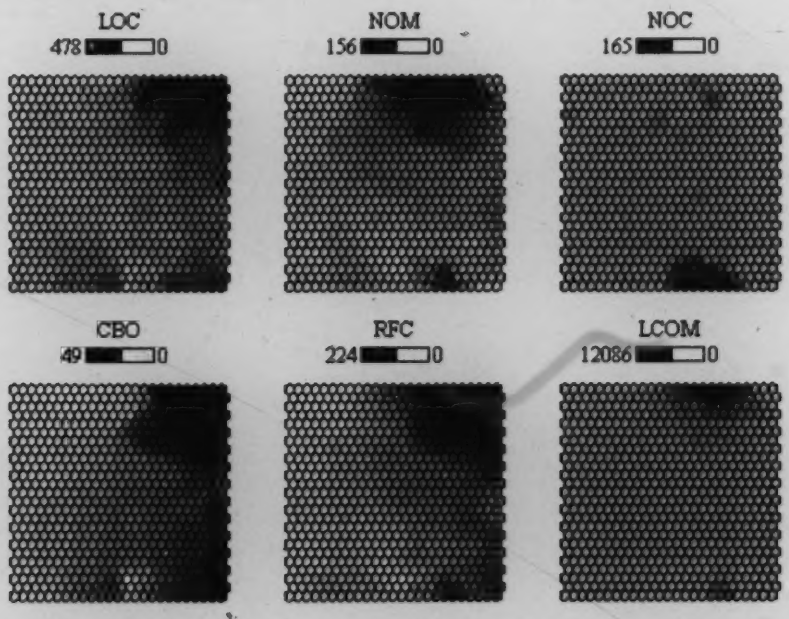


Figure 4.10 The weight maps for the Linguist data without DIT

We use the same SOM parameters as the previous experiment and train a SOM without using the DIT. Figure 4.9 and Figure 4.10 show the resulting maps. In the new clustering map that excludes the DIT, we cannot see as many distinct clusters as in Figure 4.5. There are some vaguely dark boundaries, which approximately outline a few small clusters. Combining Figure 4.9 and Figure 4.10, most software modules belong to Cluster #1, and are all small modules(with

low values in all measures). Very few software modules belong to Cluster #2, #3 and #4. These clusters correspond to the software modules with high values in some measures. Clearly, without DIT, it is impossible to cluster small modules with the remaining software measures. This experiment also suggests that for small software modules in this Linguist project, we may need to add some new software measures to differentiate them.

From the software measure data analysis on the Linguist Project, we can see the effective clustering ability of SOM. We do not have to set any assumptions on the data set as in the case of FCM. We just train the SOM on the whole data set without worrying about the outliers. From the trained SOM, we can easily distinguish clusters through the clustering map, find data popularity for each cluster from the data distribution map, and determine the software measure profile for each cluster from the weight maps. We can also find the particular software measures that have the greatest influence on the clustering map. We can train a new SOM based on some but not all of the software measures in the software project data set. Thus we may find different clustering based on different measures. Finally, we can observe the clustering effectiveness of each software measure for different types of software modules.

4.4.2 Clustering by Hierarchical SOM

Large data sets are usually organized into a hierarchical structure. For example, university students are organized into different departments, different majors and different classes. In the higher level of the hierarchy, chief managers only deal with the global issues. In the lower lever of the hierarchy, the division managers only need to be responsible for a specific part. By organizing data into a hierarchical structure, people can effectively and efficiently manage large amounts of data(Suganthan 1999, 1998). SOM is used to analyze data and give people meaningful presentations. When we train a SOM only on a small data set, such as a small software project with a few hundred modules, a single SOM may be enough. However, for a large software project with thousands of modules, it is too difficult to use a single layer SOM to analyze this data set. Therefore, to manage a large data set, we introduce HSOM(Hierarchical SOM). HSOM consists of many relatively small basic SOMs. The basic SOMs are organized into a tree-structure, which is shown in Figure 4.11. In HSOM, there is more than one layer of neurons. Data is mapped first to a parent layer and then to children layers.

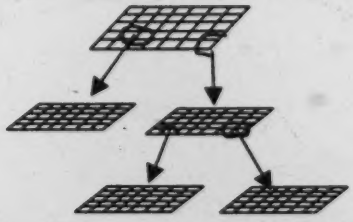


Figure 4.11 The structure of hierarchical SOM
The topmost map is the root SOM that is trained by the whole data set. The child SOM is trained from the data that are located in a certain cluster on the parent map.

4.4.2.1 Comparison with Single Layer SOM

In the single layer SOM, data are mapped to a two-dimensional map. It has only one layer of neurons. Clusters can only be identified within this map. In a hierarchically structured SOM, neurons are organized in a hierarchical structure. HSOM is trained layer by layer, rather than all on once. The major advantages of HSOM are its meaningful organization and computational efficiency.

- Meaningful organization
  One of the main tasks of SOM is clustering. In the basic SOM, highly-dimensional data can be mapped to two-dimensional map from which we can identify clusters. These clusters are all located on the same map. In the hierarchical SOM, highly-dimensional data are mapped to hierarchical maps. The clusters in this type of SOM can be found in different layers. For example, if we have a large number of universities that have 3 descriptions(country, province and city), we can use SOM to cluster them. After training using the data in Table 4.7 with basic SOM, we may get a map as shown in Figure 4.12. After training using a hierarchical SOM, we may get a map as shown in Figure 4.13.

Table 4.7 Some university data. Each row is an observation. Each data record has 3 variables: country, province and city.

University	Country	Province	City
UA(University of Alberta)	Canada	Alberta	Edmonton
UC(University of Calgary)	Canada	Alberta	Calgary
UT(University of Toronto)	Canada	Ontario	Toronto
UW(University of Waterloo)	Canada	Ontario	Waterloo
UM(University of Michigan)	USA	Michigan	Ann Arbor
MIT(Massachusetts Institute of Technology)	USA	Massachusetts	Cambridge
HU(Harvard University)	USA	Massachusetts	Cambridge
...	...	...	...

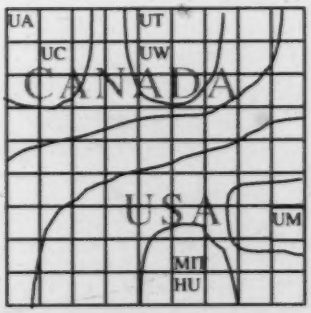


Figure 4.12 SOM for university clustering
Each university data record has 3 variable: country, province(state) and city.

From Figure 4.12, we can find some clusters from the single layer SOM. However, the relationship of these clusters is not very clear. In a real SOM clustering map, clusters are identified by gray levels, so it is very difficult to find out the hierarchical relationship of clusters. On the other hand, from Figure 4.13, the SOM grows from one 4x4 map to two 4x4 maps and finally to three 4x4 map. The root level SOM represents a rough clustering of the universities. Then for each rough cluster, child SOM is grown and finer clusters are generated. In Figure 4.13, universities are first clustered by country. Then in each country, universities are clustered by province/state. Finally, in each province/state, universities are clustered by city.

For the data that can be organized by a hierarchical structure, HSOM(Hierarchical SOM) is much more meaningful than the single layer SOM.

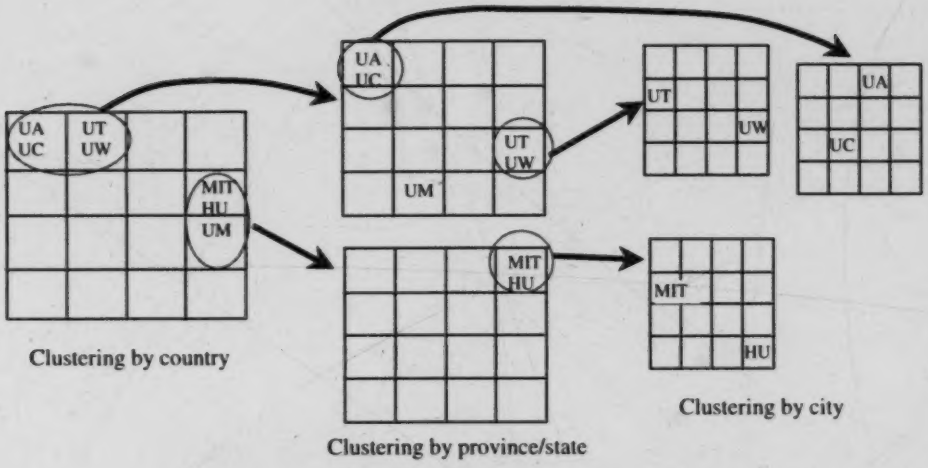


Figure 4.13 Hierarchical SOM for university clustering
The notations used in this figure are the same as in Figure 4.12.

- Computational Efficiency
  The computational complexity is exponential in the size of SOM. In one layer SOM , we have to train a large enough map at the beginning. Only after training, can we get some useful information from the maps. On the other hand, in HSOM, we first train a small root map. Once the root map is trained, we can obtain some global information about the data. Then, if we still require further details, we can select only the area that we are interested in and train a small child map. This process can be repeated until we have reached the required depth of understanding. Compared with one layer SOM, which blindly trains on all the data in one big SOM, HSOM keeps SOM small and only grows child maps on demand. Therefore, HSOM can greatly improve the training speed.

In HSOM, we can easily build a cluster hierarchy. However, it is difficult to know the distance between two clusters in different SOMs. In other words, it is not easy to laterally compare clusters in different child SOMs. On the other hand, in single layer SOM, all the clusters are displayed on one big map. The relative distance between clusters indicates their relative level of similarity. This is perhaps the major disadvantage of HSOM.

4.4.2.2 Clustering of the JDK data by HSOM

The JDK data set has 3016 records. In order to build a hierarchical SOM, we keep the size of SOM small. We keep using 10 x 10 SOM in all maps in the hierarchy. For the root SOM, the training parameters are listed below:

- Size of the maps: 10 x 10
- Normalization of variables: logistic normalization
- Initial neighborhood radius: 3
- Initial condition: randomly initiated connections
- No frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 1000 iterations

Once the SOM is trained, the maps are as shown in Figure 4.14 and Figure 4.15. According to the clustering map, this large data set has been clustered into roughly 4 clusters: A, B, C and D. The popularity of each cluster can be determined from the data distribution map. The characteristics of each cluster(profile of each cluster) can be determined from the weight maps.

From the "root" SOM, we capture the global clustering information and ignore the detailed clustering information. This means that we can use this "root" SOM to do some high level cluster analysis. We now examine the features of these high level clusters. Cluster-A is the largest cluster. It has low values in all software measures except DIT, which has quite different values inside Cluster-A. Cluster-B has middle values in all software measures except DIT, which has low values in Cluster-B. Cluster-C has much fewer data points than Cluster-A. It has high values in LOC, DIT, CBO and RFC, but has

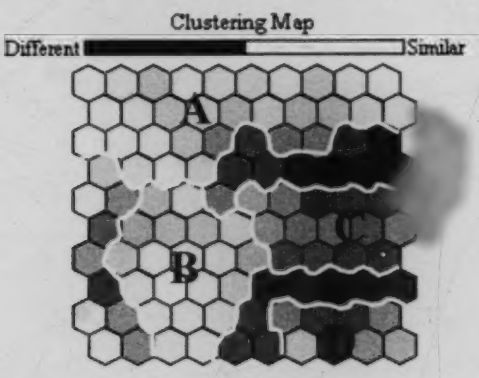


Figure 4.14 The clustering map for the JDK data

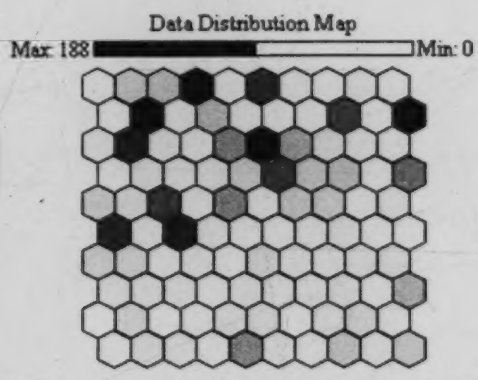


Figure 4.15 The data distribution map for the JDK data

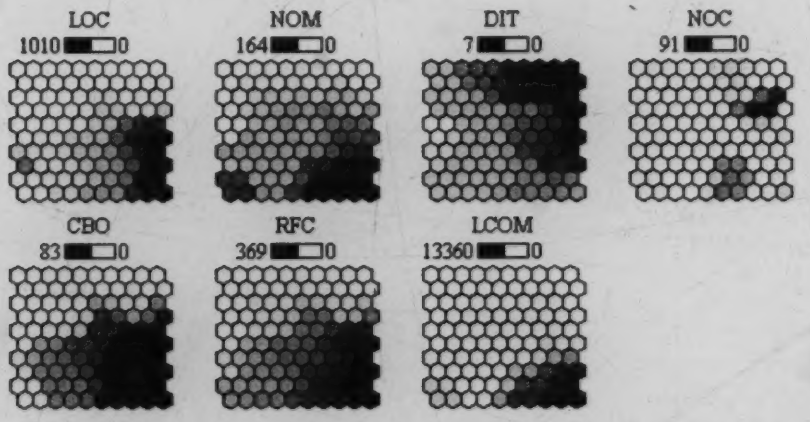


Figure 4.16 The weight maps for the JDK data

middle values in NOM, and low values in NOC and LCOM. Cluster-D also has much fewer data points than Cluster-A. It has high values in LOC, NOM, CBO, RFC and LCOM, but has low values in DIT and NOC. Because this SOM clustering is at such a high level, the software modules in each cluster have heterogeneous names. Therefore, we grow child SOMs from this "root" SOM, and get finer clustering.

As an example, we grow a child map from Cluster-B, which has 496 data records. We use a 10 x 10 SOM for this child SOM. The training parameters is the same as in the root SOM. However in this child SOM, we do not use all the software measures to train. The root SOM uses 7 measures: LOC, NOM, DIT, NOC, CBO, RFC and LCOM. This child SOM only uses 5 measures: LOC, NOM, CBO, RFC and LCOM. The two measures DIT and NOC are removed from training. If we examine the histograms of the two data sets used by the root SOM and child SOM, we can find out why we must remove DIT and NOC. Figure 4.17 and Figure 4.18 show these histograms.

In Figure 4.17, although the histograms are skewed, all the software measures have multiple values, so we can use all the measures to train the root SOM. In Figure 4.18, all the histograms except DIT and NOC become flatter. 99% of software modules in Cluster-B have DIT = 1, and 93% of software modules in Cluster-B have NOC = 0. This high commonality over DIT and NOC valuse is why the root SOM put these software modules into one distinct cluster(Cluster-B). However, this high commonality of DIT and NOC makes further training very difficult, both linear and logistic normalization will not work over DIT and NOC. Therefore, we simply remove these two measures, because they will not contribute further to clsutering.

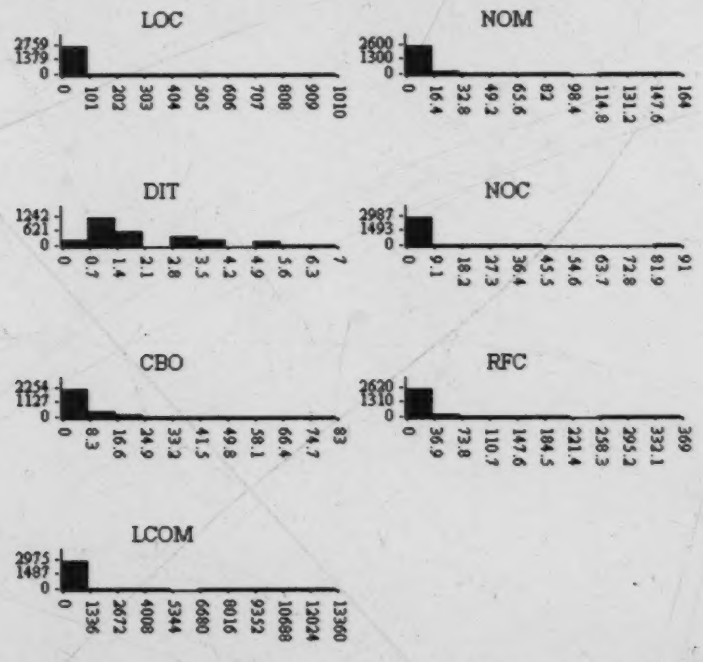


Figure 4.17 Histograms of all software measures for all the modules in the JDK data set

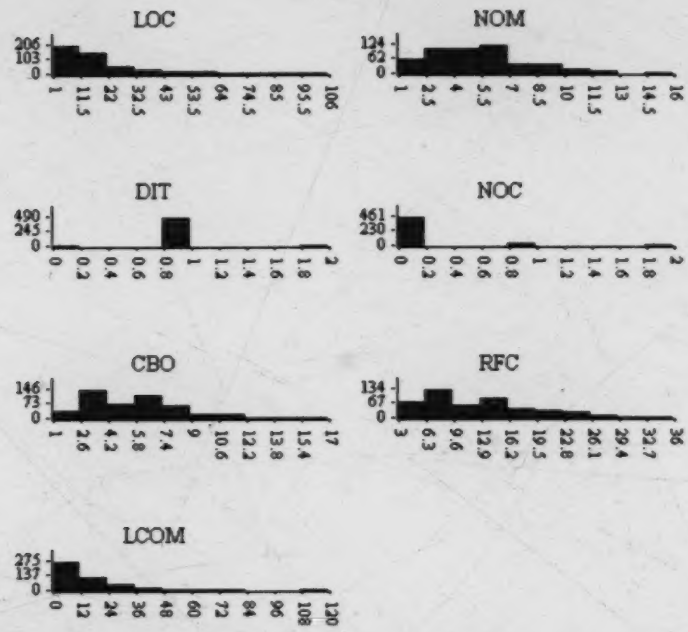


Figure 4.18 Histograms of all software measures for the modules in Cluster-B

After training without DIT and NOC, the resulting maps are shown as in Figure 4.19, Figure 4.20 and Figure 4.21. In this child SOM, the software modules(Java classes) in Cluster-B is further clustered. We can see that in the root SOM, Cluster-B seems to be a homogeneous cluster, and represents those middle sized modules. However, once we train using the data in Cluster-B with a child SOM, we find quite different clusters inside Cluster-B.

Combining the clustering map in Figure 4.19 and the data distribution map in Figure 4.20, we can find approximately 5 non-trivial clusters. The 5 clusters include approximately 412 data records, or 84% of the 496 data records. The other 84 data records are unclassified because they are located in the cluster boundaries and do not fit into any cluster. In order to determine the profile of each cluster, we combine the clustering map with the weight maps. Then we can get the descriptive profiles for each cluster. In Table 4.8 we use some linguistic descriptions such as "high" and "low" to decribe SOM clusters. For example, if the neurons of a cluster in the LOC weight map has bright color, then the data records that are located in this cluster will have high LOC values. If the neurons of a cluster in the LOC weight map has dark color, then the data records that are located in this cluster will have low LOC values.

However, these descriptions are quite subjective because they just correspond to the intuition of the observer. In Chapter 6, we will discuss the application of sets and fuzzy sets to get objective linguistic descriptions for SOM clusters.

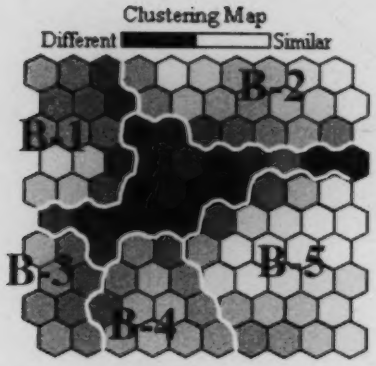


Figure 4.19 The clustering map for the JDK data in Cluster-B

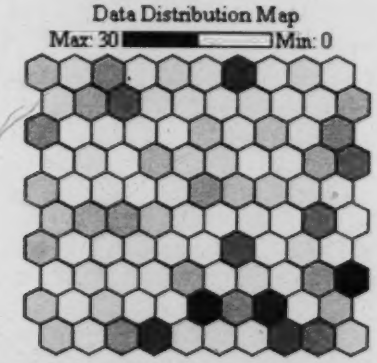


Figure 4.20 The data distribution map for the JDK data in Cluster-B

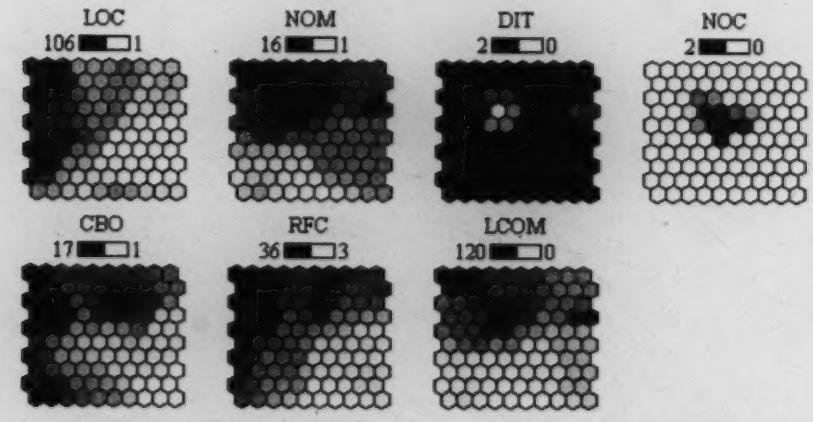


Figure 4.21 The weight maps for the JDK data in Cluster-B

Table 4.8 Cluster profiles for the 5 clusters in Figure 4.19

Cluster	#Data Records	LOC	NOM	CBO	RFC	LCOM
B-1	46	high	high	high	high	high
B-2	116	middle	middle	middle	middle	middle
B-3	35	high	low	middle	middle	low
B-4	72	middle	low	middle	middle	low
B-5	143	low	middle	low	low	middle

Table 4.9 Descriptions of SOM clusters based on the software module names

Cluster	Major Class Names	Descriptions
B-1	"...Helper", "...Iterator"	Most classes in this cluster are helper class, or iterator class.
B-2	"...Helper", "...Entry"	Most classes in this cluster are helper class, or deal with entry functions, such as key entries.
B-3	"...Handler", "...Icon"	Most classes in this cluster are handler, such as exception handler. Some classes are related to icon operations.
B-4	"...Handler", "...Listener", "...Loader"	Some classes in this cluster are handlers such as exception handler. Some classes are listeners such as event listener. Some classes are loader class, responsible for loading resources.
B-5	"...Holder", "...Icon", "...Handler"	Most classes in this cluster are holder classes, which hold some memory for later use. Some classes are related to icon operations. Some classes are handlers, such as exception handler.

According to Table 4.8, different clusters have distinct profiles. This is why the child SOM can distinguish them.

This child SOM gives us a finer clustering for the data records in Cluster-B. Now if we examine the clusters, the names of software modules in each cluster are much more homogeneous than in the case of Cluster-B. We can get some texual descriptions for each cluster based on the name of software modules. These texual descriptions are shown in Table 4.9.

We have seen that the child SOM can help us find detailed cluster information from the root SOM. From child SOM, we still can grow a new SOM, which can be called a grandchild SOM. As an example, we can grow a grandchild SOM from Cluster-B-5 in the child SOM. Before training a grandchild SOM, let's first check the histograms for the data in Cluster-B-5. If any measures only have single values, we must remove them.

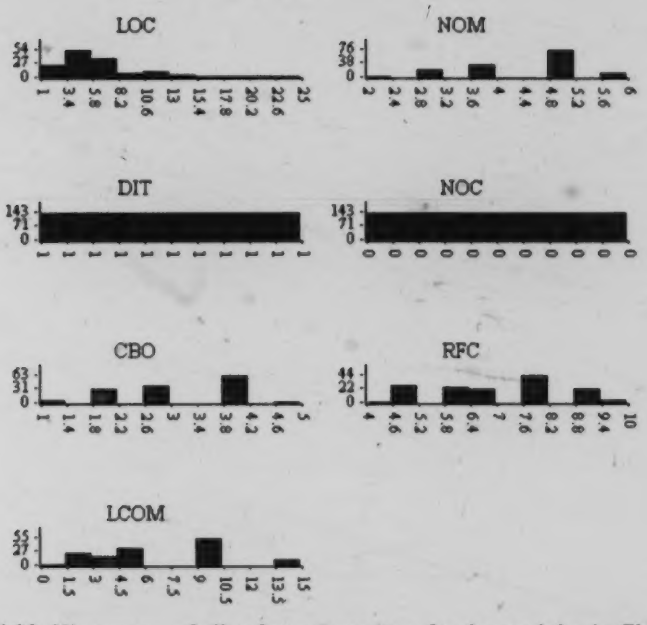


Figure 4.22 Histograms of all software measures for the modules in Cluster-B-5

According to the above histograms, only DIT and NOC need to be removed, because they only have a single value over all the data in Cluster-B-5. Let's examine the training parameters for the grandchild SOM after removing DIT and NOC.

There are 143 data records in Cluster-B-5. We still use a 10 x 10 SOM. The parameters of the grandchild SOM are listed below:

- Size of the maps: 10 x 10
- Normalization of variables: logistic
- Initial neighborhood radius: 3
- Initial condition: randomly initiated connections
- Frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 1000 iterations

Note that this grandchild SOM has a different parameter from its parent and grandparent SOMs. This grandchild SOM uses frequency sensitive learning, while its parent and grandparent SOMs do not. In the root SOM and the first child SOM, we map 3016 and 496 software modules to 10x10 SOMs respectively, so it is very unlikely to have idle neurons(see detailed discussion in section 3.6.3). Thus we do not need to apply frequency sensitive learning to them. Now, this grandchild SOM only has 143 input data records(from Cluster-B-5), so it is possible to generate idle neurons(see detailed discussion in section 3.6.3). Therefore, we apply frequency sensitive learning to this new grandchild SOM.

After learning, we get a collection of maps, as shown in Figure 4.23 to Figure 4.25. According to these maps, the data records that are already fairly homogeneous are further clustered. In the clustering map in Figure 4.23, we can find 6 clusters. The prototypes, which are the averages of

the data records in a cluster, are listed in Table 4.10 for each cluster. The 6 clusters include approximately 120 data records, or 84% of the 143 data records. The other 23 data records are unclassified because they are located on the cluster boundaries and do not fit into any cluster.

Table 4.10 Cluster prototypes for the 6 clusters in Figure 4.23

Cluster	#Records	LOC	NOM	DIT	NOC	CBO	RFC	LCOM
B-5-a	34	5	5	1	0	4	9	10
B-5-b	7	6	6	1	0	2	6	15
B-5-c	28	9	3	1	0	3	6	3
B-5-d	20	2	5	1	0	2	5	10
B-5-e	19	5	5	1	0	4	8	4
B-5-f	12	4	4	1	0	2	6	6

Figure 4.23 The clustering map for the JDK data in Cluster-B-5

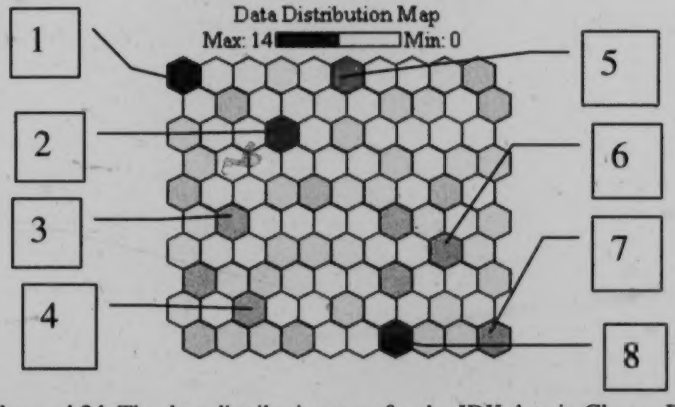


Figure 4.24 The data distribution map for the JDK data in Cluster-B-5
Some "hot" neurons(highly homogeneous clusters) are labeled.

If we examine the data distribution map in Figure 4.24, we can find some "hot" neurons. These are highly homogeneous clusters. Some data records are so similar that they can not be distinguished by different neurons. These highly homogeneous software module clusters should be highlighted by software managers and developers, because they are very likely to demonstrate similar behavior. Table 4.11 lists the software modules that are located in these "hot" neurons.

Table 4.11 The "hot" neurons(highly homogeneous clusters) in Figure 4.24

Neuron ID	# Classes	Description
1	12	All the classes have the same software measure values: LOC=6, NOM=5, DIT=1, NOC=0, CBO=4, RFC=8, LCOM=10 Classes here have names "...Holder" and are inherited from CosNaming. Classes in this cluster function as holder class.
2	8	All the classes have the same software measure values: LOC=5, NOM=5, DIT=1, NOC=0, CBO=4, RFC=9, LCOM=10 Classes have names "...Holder" and are in CORBA package. Classes in this cluster function as holder class.
3	4	All the classes have the same software measure values: LOC=12, NOM=4, DIT=1, NOC=0, CBO=3, RFC=8, LCOM=6 Classes have names "...Border". Classes in this cluster manage the drawing of borders.
4	4	All the classes have the same software measure values: LOC=9, NOM=3, DIT=1, NOC=0, CBO=3, RFC=6, LCOM=3 Classes have names "...Resource". Classes in this cluster manage resource.
5	6	All the classes have the same software measure values: LOC=5, NOM=5, DIT=1, NOC=0, CBO=3, RFC=9, LCOM=10 Classes have names "...Holder" and are in CORBA package. Classes in this cluster function as holder class.
6	5	All the classes have the same software measure values: LOC=2, NOM=5, DIT=1, NOC=0, CBO=2, RFC=5, LCOM=10 Classes have names "...Icon". Classes in this cluster are related to icon drawing, moving and updating.
7	5	All the classes have the same software measure values: LOC=3, NOM=4, DIT=1, NOC=0, CBO=2, RFC=6, LCOM=6 Classes have names "...Icon". Classes in this cluster are related to icon drawing, moving and updating.
8	14	All the classes have the same software measure values: LOC=5, NOM=5, DIT=1, NOC=0, CBO=4, RFC=8, LCOM=4 Classes have names "...Holder" and are inherited from CORBA. Classes in this cluster function as holder class.

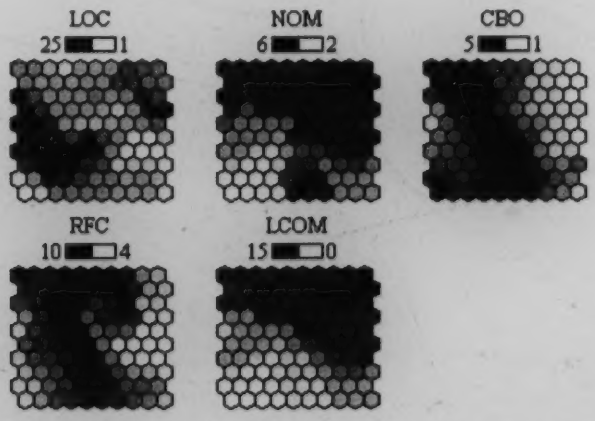


Figure 4.25 The weight maps for the JDK data in Cluster-B-5

4.4.2.3 Summary

From the above discussion, we see how a SOM hierarchy can be built from the root to the branches. Figure 4.26 shows one path of the hierarchy for the JDK data set. For the JDK data set, which has 3016 data records, if we use single layer SOM, we need at least a 50 x 50 SOM to represent the detailed cluster information. For a SOM of this size, the computational complexity is quite high, and in the final map all the detailed information is shown simultaneously, which makes it difficult to read. Using hierarchical SOM, we first train a small SOM and capture the most important global cluster information of this data set, then we grow child SOMs only on those clusters in which we are interested. In this way, we can get finer and finer clusters. Organizing SOMs hierarchically is more meaningful, because we can get different levels of clustering from different levels of SOMs. According to the experiment on the JDK data set, the first level(root) SOM only roughly differentiates all the software modules into such clusters as small size, middle size and large size. In the root SOM, the clusters are so rough that we can not find any common name descriptions for the software modules for each cluster. Then in the child SOM, middle sized modules are further differentiated into many different clusters, and we can find some meaningful name descriptions of software modules for each cluster. Finally, when we grow grandchild SOMs from child SOMs, software modules are distinguished into more homogeneous clusters. In this grandchild SOM, software modules in each cluster have almost identical software measure values. We can find some highly homogeneous clusters(hot neurons), which help us to identify similar software modules. According to the name descriptions of these software modules, the software modules in one hot neuron have very similar functions and behaviors.

From the JDK data experiment, we can see that the process of building hierarchical SOMs is a highly interactive process. It requires a lot of user's interaction. The user must select the neurons from which to grow a new SOM, and then repeat this procedure. For practical application of hierarchical SOM data exploration, this user interaction must be implemented by a friendly user interface. In Chapter 7, we will discuss the design of an interactive SOM tool for software measure analysis.

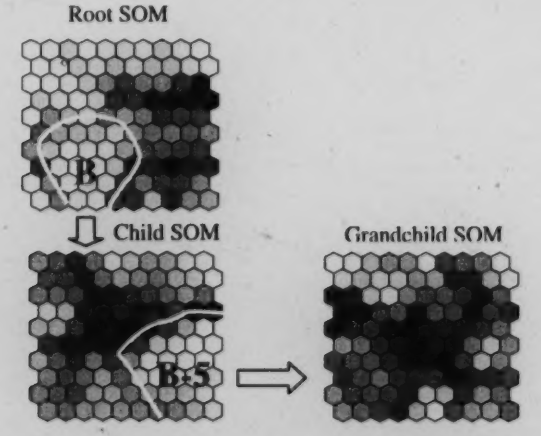


Figure 4.26 Building hierarchical SOM
First using all the data to build root SOM, then take the data in Cluster-B as input data to train the child SOM, finally take the data in Cluster-B-5 as input data to train the grandchild SOM.

5 Using Different Similarity Measures in SOM

In quantitative software engineering, we often need to compare the similarity between software modules. For example, software managers can put similar modules together and make many different module groups, then they dedicate different software developers to different module groups. In this way, developers can concentrate on a specific problem in a specific field, and will be more efficient and productive. In addition, because similar software modules tend to exhibit similar behavior, once we find a design defect in one module, we can easily go to other modules in the same group and fix the problem before failure occurs. The question is how to determine whether or not two software modules are similar. When we use SOM to do clustering, we must compare the input data with the neuron weight vector to determine how similar they are. Therefore, a similarity measure must be defined before running SOM, and this similarity measure must be suitable to the particular problem. In the following sections, we discuss a different similarity measure and its application to SOM.

5.1 A Neural Model for Similarity Measure

In quantitative software engineering, we use software measures to describe software modules. Once we know the software measures of two software modules, how do we compare their similarity? People usually use Euclidean distance to compare the similarity between two software modules. If the Euclidean distance between two software measure vectors is small, the two software modules are considered to be similar. However, because software consists of highly complex entities, no one knows if Euclidean distance is the best measure of similarity for software measures. People use Euclidean distance just because it is the most widely and frequently used distance function.

In order to deal with this problem, we can use a new similarity(proximity) measure based on neural networks(Pedrycz, 2001b). Because a neural network is a universal function emulator(Rojas, 1996), we can use it for an unknown "distance function". We can use the existing knowledge to train the neural network so that it will act as a distance function that is customized to fit the specific problem space. Using this new similarity measure, we can make the SOM training more suitable to the characteristics of a specific software project. Figure 5.1 shows the architecture of a SOM based on neural network similarity measure.

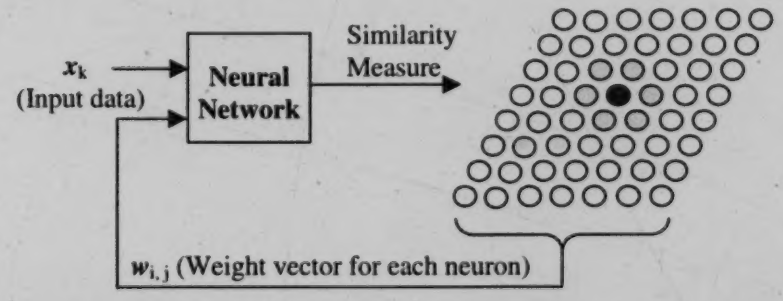


Figure 5.1 SOM training architecture based on neural network similarity measure

Before the SOM training, the neural network must be trained using supervised learning. The neural network can be a simple one-hidden-layer network, with one output as the similarity measure. The input is a pair of data records. The target should be the known similarity output, which represents our existing knowledge. The basic neural network structure is shown in Figure 5.2.

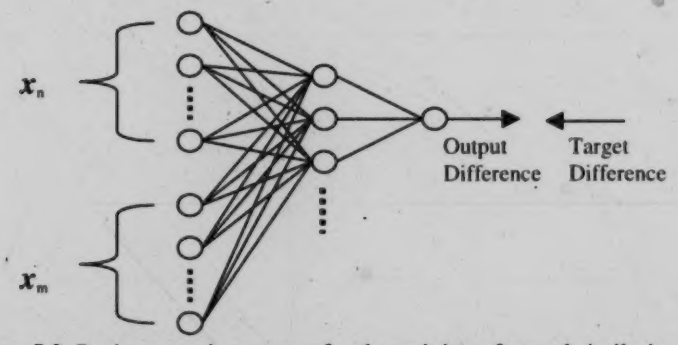


Figure 5.2 Basic network structure for the training of neural similarity measure
x_n and x_m are two input data records, n=1, 2...K, m=1, 2...K
(K is the total number of input data records)

5.2 *Experiment with Synthetic Data*

In order to highlight the need for this neural network similarity measure, we first experiment with a synthetic data set. In this experiment, we can see the obvious improvement of SOM using this neural network similarity measure.

5.2.1 Description of Experimental Data

Consider a 5-D fuzzy XOR(exclusive-or) problem where the data can be described by the following function(Pedrycz, 2001b):

$$F: [0,1]^5 \rightarrow [0,1]$$
$$\text{such that } F(\boldsymbol{x}) = F(x_1, x_2, \ldots, x_5) = x_1 \oplus x_2 \oplus \ldots \oplus x_5 \qquad (5.1)$$
$$x_1 \oplus x_2 = (\bar{x}_1 t x_2) s (x_1 t \bar{x}_2) \quad \text{with } \bar{x} = 1 - x$$

The t-norm and s-norm are realized in the form of the product and probabilistic sum(Pedrycz, 1998). In this synthetic data set, we want to find an output similarity measure using the input data. In other words, if we have two input records: x_n and x_m, we want to represent a "distance function" $D(x_n, x_m)$, such that if $D(x_n, x_m)$ is small, then $|F(x_n) - F(x_m)|$ is small, otherwise $|F(x_n) - F(x_m)|$ is large.

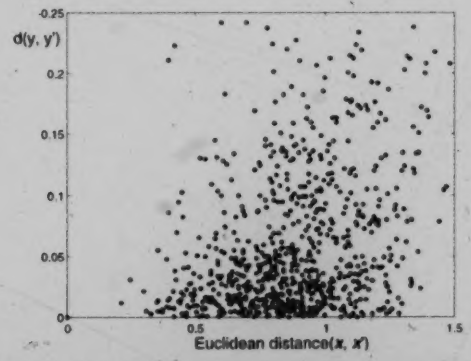


Figure 5.3 Scatter plot of distances in the input space and output space
$d(y, y') = |y - y'| = |F(x) - F(x')|$
(40 data points that give rise to 1600 pairs of data were used in this plot)

If we use Euclidean distance for $D(x_n, x_m)$, we can plot the relationship between the Euclidean distance in input space and for the distance in output space($|F(x_n) - F(x_m)|$). In Figure 5.3, we see a high level of scattering. It is clear that the same distance value in input space(X axis) corresponds to many quite different distance values in the output space(Y axis). This means that using Euclidean distance as the similarity measure for this synthetic data set is not appropriate. It will not give us any hint on the distance of output space.

5.2.2 Supervised Learning in Neural Networks

Now let's use the neural network similarity measure. The basic neural network structure is shown in Figure 5.2. It has one hidden layer with 15 neurons. The Sigmoid function is used as the neural nonlinear function. The standard back-propagation algorithm is used with a learning rate of α=1 to train this neural network. The training process is monitored by the performance index Q, which is the sum of squared errors between the Target and the actual Output of network. Figure 5.4 shows the change in Q during learning. We can see that the learning is smooth and the value of Q converges rapidly.

Once the neural network is trained, we plot the relationship between the neural network similarity and the output distance using the same input data as in Figure 5.3. Figure 5.5 shows this plot. We can see the obvious association between the neural network similarity measure and the output distance. They are much more consistent than using Euclidean distance. Therefore, expressing similarity by neural networks can gives us a better result for this problem.

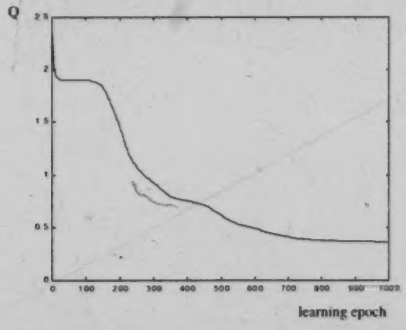


Figure 5.4 Performance index Q in successive learning epochs

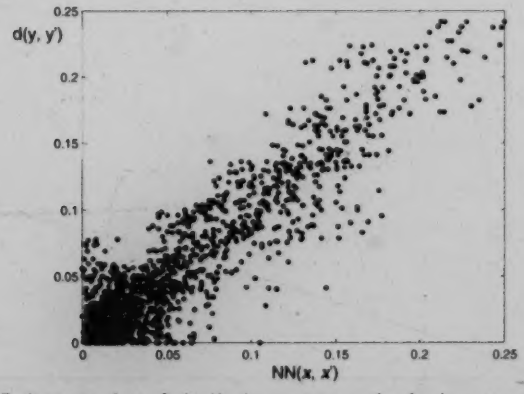


Figure 5.5 Scatter plot of similarity measures in the input and output space
$NN(x, x')$ denotes the neural network similarity measure
$d(y, y')$ is the difference in output space, $d(y, y') = |y - y'| = |F(x) - F(x')|$

5.2.3 Unsupervised Learning in SOM

Having built a neural similarity measure we can apply it to SOM. We use the same input data(40 points) as in Figure 5.3 and Figure 5.5. We train a SOM with the following features:

- Size of the maps: 6 x 6
- Initial condition: randomly initiated connections
- No frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 1000 iterations

Notation:
6
0.52-0.61
means:
there are 6 data points for this neuron. The fuzzy XOR values for these data points range from 0.52 to 0.61.

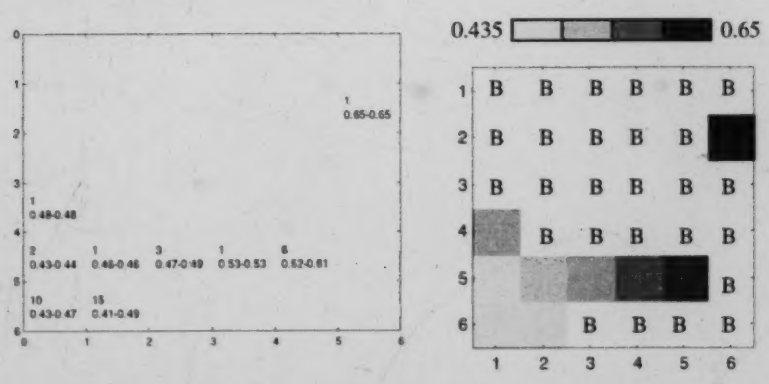


Figure 5.6 Visualization of fuzzy XOR data on the SOM constructed using the neural network similarity measure. The map on the right is the gray level representation of the map on the left. The gray level is the average fuzzy XOR value of all data points for a given neuron. A neuron marked as "B" indicates that there are no data points mapped to it.

Notation:
2
0.45-0.46
means:
there are 2 data points for this neuron. The fuzzy XOR values for these data points range from 0.45 to 0.46.

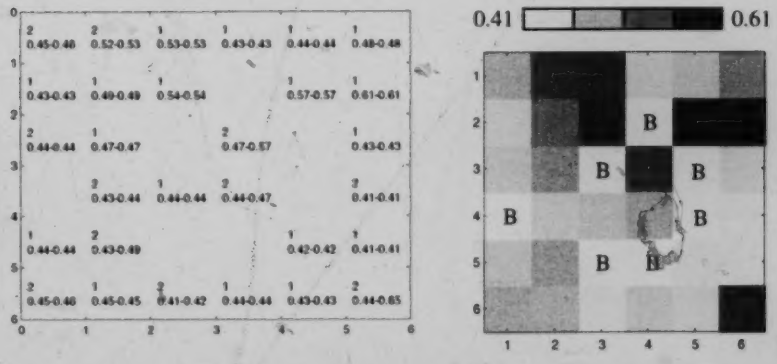


Figure 5.7 Visualization of fuzzy XOR data on the SOM using Euclidean distance. The map on the right is the gray level representation of the map on the left. The gray level is the average fuzzy XOR value of all data points for a given neuron. A neuron marked as "B" indicates that there are no data points mapped to it

After the SOM is sufficiently trained, we map all the input data back onto the SOM, as shown in Figure 5.6. For comparison, we also train a SOM with the same size but using Euclidean distance as the similarity measure, and show it in Figure 5.7.

In Figure 5.7, the data points are scattered all over the map. Data points with similar output do not locate in neurons which are close to each other. This Euclidean based SOM does not give us any cluster information in terms of output values. On the other hand, in Figure 5.6, data points with similar output are put in the same or close area on the map. The input data are clustered by SOM in terms of their output values.

From this experiment with synthetic data, we can see the obvious improvement in SOM using neural similarity measure. For a data set with complex relations between input and output, Euclidean distance is not a reliable output similarity measure, instead, a neural similarity measure can play the role to associate output similarity with input data. In quantitative software engineering, the relationship between internal software measures and external software behaviors are very complex, so we can apply this neural model as a similarity measure between software modules.

5.3 Experiment with Software Measure Data

We use the MIS data set(Munson, 1996) to demonstrate the application of a neural network similarity measure in quantitative software engineering. The detailed description of the MIS data set can be found in section 4.2.1. There are 390 modules in the MIS data set. Each module is characterized by 11 software measures. Each module has a label, "Changes", which is the number of modifications made to this module during development and maintenance. Here we define similarity between two software modules as having similar "Changes". "Changes" corresponds to the faults in a software module, so having similar "Changes" implies having similar complexity, which is the main cause of faults. Our goal is to find a neural network so that if we input two MIS data records, it will output a similarity number and this number will reflect the difference in their "Changes". Finally we can train a SOM based on this new similarity measure.

5.3.1 Supervised Learning in Neural Networks

The neural network is shown in Figure 5.2. There are 15 hidden layer neurons. x_n and x_m are two arbitrary input records, each of which has 11 variables corresponding to 11 software measures. They form an input pair, so there are 22 input neurons. We pick 50 data points from the MIS data for supervised learning, so there will be 2500 data pairs. The target is the difference between the "Changes" of the two input records. The performance index Q is monitored during training, as shown in Figure 5.8. According to the learning curve, the neural network converges smoothly.

Once the neural network is trained, we can plot the relationship between neural similarity measure and difference in "Changes", as shown in Figure 5.9. The relationship between Euclidean distance and difference in "Changes" is plotted in Figure 5.10 for comparison. Obviously, neural similarity measure shows a strong linear association with difference in "Changes". On the other hand, in Figure 5.10, data pairs are scattered. There is no obvious association between Euclidean distance and difference in "Changes". In other words, in case of Euclidean distance, there is no obvious association between external behavior("Changes") and internal attributes(the 11 software measures). In case of the neural network similarity measure, there is strong association.

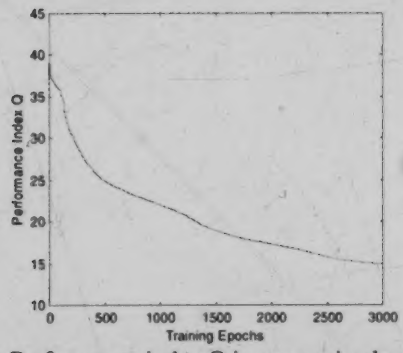


Figure 5.8 Performance index Q in successive learning epochs

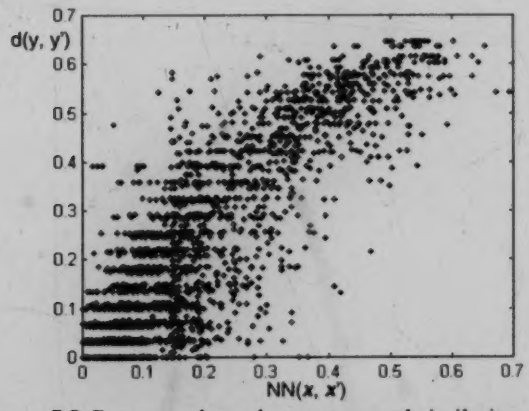


Figure 5.9 Correspondence between neural similarity measure
and difference in "Changes"(Normalized) for 2500 MIS data pairs.
d(y, y') is the normalized difference between the "Changes" for two data records.
NN(x, x') is the neural similarity number for the two data records.

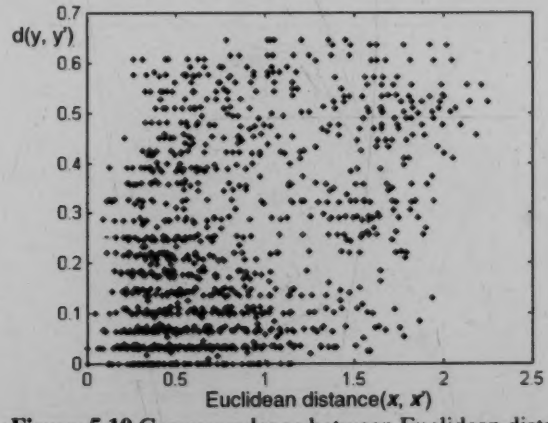


Figure 5.10 Correspondence between Euclidean distance
and difference in "Changes"(Normalized)) for 2500 MIS data pairs.
d(y, y') is the normalized difference between the "Changes" for two data records.

5.3.2 Unsupervised Learning in SOM

Now we apply this neural similarity measure to SOM. A 6 x 6 SOM is used for both Euclidean distance and neural similarity measure. The SOM has the same parameters as the one used in section 5.3.3. The 50 MIS data records are trained using SOM. When the SOM become stable(sufficiently trained), we plot the data distribution map for both Euclidean distance and neural similarity measure, as shown in Figure 5.11 and Figure 5.12.

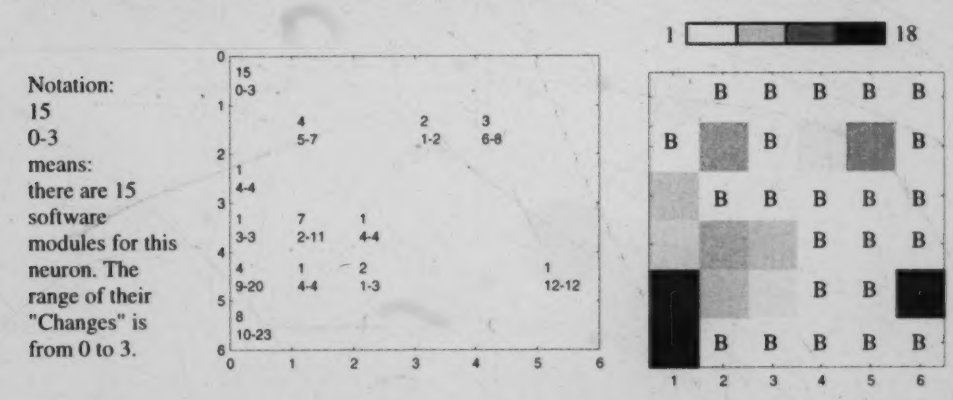


Figure 5.11 The SOM using neural similarity measure for 50 MIS data records. The map on the right is the gray level representation of the map on the left. The gray level is the average of "Changes" of all software modules for a given neuron. A neuron marked as "B" indicates that there are no data points mapped to it.

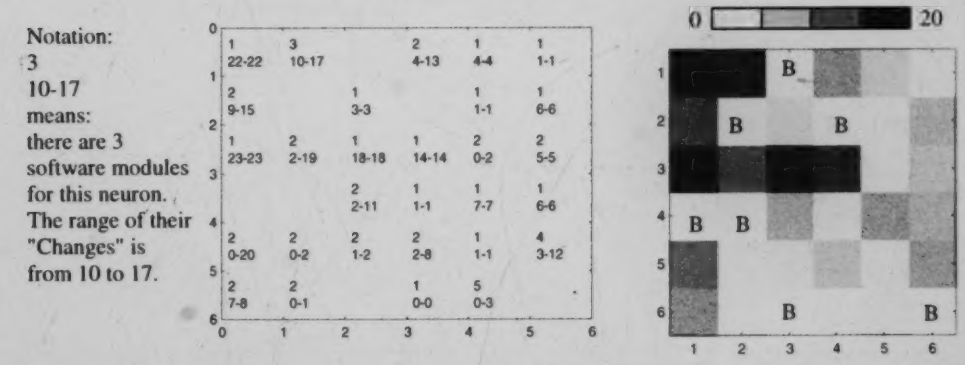


Figure 5.12 The SOM using Euclidean distance for 50 MIS data points. The map on the right is the gray level representation of the map on the left. The gray level is the average of "Changes" of all software modules for a given neuron. A neuron marked as "B" indicates that there are no data points mapped to it.

We can see that for the SOM based on Euclidean distance for similarity measure, that data is scattered all over the map, which makes it difficult to identify clusters. For the SOM based on neural similarity measure, there is not so much scattering. This low level of scattering makes it easy to identify clusters from SOM.

5.4 *Summary*

In the above sections, we discussed a neural model to express similarities. We demonstrated the advantages of neural network similarity measure, and the limitations of conventional distance functions, such as Euclidean distance. The neural similarity measure is generated from a specific problem space and is more suitable for this problem space than "general-purpose" distance functions. In software engineering, software objects are highly complex and it is difficult to define a general distance function to measure the similarity between them, so the neural network similarity measure applies well in this field. Once the neural similarity measure is acquired(neural network is trained), we can easily apply it to SOM. Because the essence of SOM is to organize data based on their similarities, this neural similarity measure can improve the SOM performance to fit the specific problem space.

6 Granular Description of SOM

In this chapter, we discuss the granular description of SOM. SOM can help us detect clusters in a data set. Granular descriptions can help us describe each cluster in a way that is easy to understand by human. We can use sets and fuzzy sets to get linguistic descriptors of a cluster in SOM. Applying granular description, we can make SOM become a more user-friendly data-mining tool.

6.1 Motivation

In Chapter 4, we discussed software measure analysis using SOM. We have shown how to identify clusters using the clustering map and how to determine cluster characteristics using the weight maps graphically. We have also discussed using prototypes(see discussion of prototypes in section 4.4.2) to indicate the cluster characteristics numerically. However, the graphical description of clusters(by the weight maps) is not convenient to record, and is so subjective that different user may obtain different description. Also, the numerical prototypes, although precise and objective, are too abstract to understand easily.

In our daily life, we describe things using words. For example, we describe brightness by "Dark" vs. "Bright". On the other hand, we could use Watts per Square Meter to describe the brightness. Obviously, the latter descriptor is much more difficult to understand. In software measure analysis by SOM, once we know the cluster, we can describe the software modules in this cluster in such words as "Large LOC" and "Small DIT". We could also describe those modules numerically as "LOC = 450" and "DIT = 1". Obviously, the words give us more understandable and meaningful descriptors(Zadeh, 1996). Figure 6.1 shows the two types of descriptors.

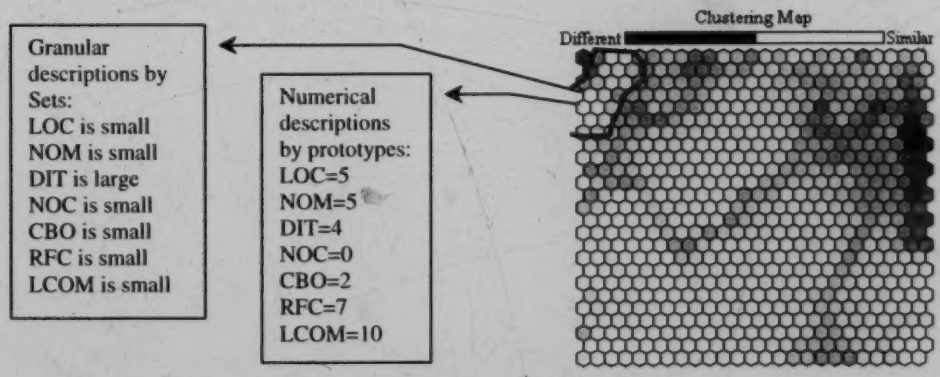


Figure 6.1 Comparison between granular descriptors and numerical descriptors for a cluster in SOM

The most important feature of SOM analysis is its potential as a very user-friendly visualization tool. This feature is achieved by using a collection of maps, rather than numbers. Now, when we describe clusters in SOM, we still want to make the descriptors user-friendly, so we apply granular descriptors to SOM clusters.

Granular descriptors include sets and fuzzy sets descriptors. In the following sections, we use both of them to describe the characteristics of SOM clusters.

6.2 *Using Sets as Granular Descriptor of SOM*

The concept of sets is to help people organize, summarize and generalize knowledge about objects. A set is a collection of some objects with a sharp boundary. Inside a set, all the members share some general properties(Pedrycz, 1998). For a given set A(A is in universe X) and a given object x($x \in X$), x is either inside the set A, or outside A. If we use 1 for objects inside the set A and 0 for objects outside, then a membership function $A(x)$ can be written as:

$$A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A \end{cases} \qquad (x \in X) \tag{6.1}$$

In our problem, we want to use sets to describe the clusters in SOM. Because software measures are highly-dimensional, we must construct sets for each variable. For example, Figure 6.2 shows some constructions of sets for software measures.

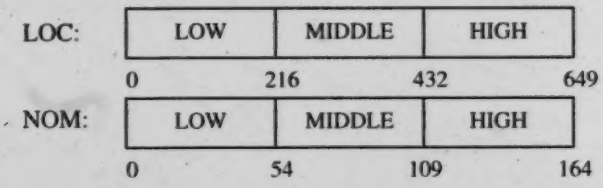


Figure 6.2 Sets construction for software measures LOC and NOM

In the above sets constructions, the maximum number of LOC is 649 and maximum number of NOM is 164. We divide the ranges evenly into three sub-ranges. Each sub-range corresponds to a set, such as "LOW", "MIDDLE" and "HIGH". Once we have constructed the sets, we can use them to describe software modules. For example, if a software module has LOC = 250 and NOM = 15, then we can describe it as having *middle* LOC and *low* NOM.

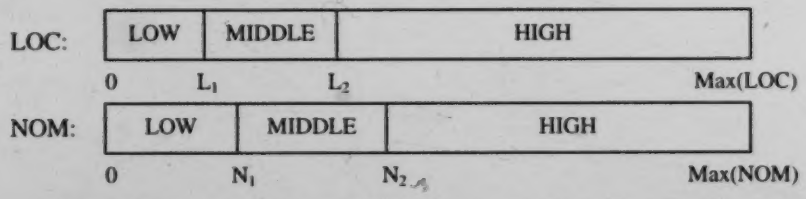


Figure 6.3 Construction of equalized sets for a software measure data set
Max is the maximum value of a software measure. If the number of modules in this software project is N, then there are N/3 modules with LOC values between 0 to L_1, with NOM values between 0 to N_1. There are N/3 modules with LOC values between L_1 to L_2, with NOM values between N_1 to N_2. There are N/3 modules with LOC values between L_2 to Max(LOC), with NOM values between N_2 to Max(NOM).

In the construction of the sets in Figure 6.2, the sub-ranges are evenly divided over the whole range. This is fine for those data that have flat histograms. In case of the software measure data set, the histograms are usually skewed to the low end, as shown in Figure 4.8. So if we still use the set construction in Figure 6.2 on the software measure data set, we will see that most software modules are in the set labeled "LOW" and only a few modules are in the set labeled "MIDDLE" and "HIGH". Therefore, for software measure data sets, we use another approach to construct sets, as shown in Figure 6.3. In Figure 6.3, in each sub-range, the number of software modules is

identical. If we want to construct multiple sets, say c, we will have c sub-ranges. Then we can use the construction approach as shown in Figure 6.3 and simply make each sub-range have N/c software modules(N is the total number of modules in the software project).

Once the sets are constructed, we can proceed to describe clusters in SOM. In SOM, we can identify clusters using the clustering map and the data distribution map. For each cluster, we can list all the data records that are located in this cluster. Then we use the following algorithm to find set descriptors for this cluster.

1. Combine all the sets of all the variables to make c^n associations where c denotes the number of sets and n denotes the number of variables in the data set.
2. For each data record $x_i (i = 1, 2 \ldots N_{cl}$, N_{cl} denotes the number of data records in this cluster), analyze the c^n associations and find an association s_i that fits x_i.
3. Find the dominant association s_d from all $s_i (i = 1, 2 \ldots N_{cl})$
4. Calculate the descriptor coverage that is the percentage of data records described by s_d

In the above algorithm, each association is a tuple in the Cartesian product of all sets and variables. For example, in Figure 6.3, one possible association could be written as "LOC is LOW and NOM is MIDDLE". If there are c sets and n variables, the total number of associations is c^n, which could be a very large number if c or n is large. Therefore we must consider the computational cost when using large c or n. In the second step of the above algorithm, if the value of each variable in x_i is inside the corresponding set in association s_i, then we say the association s_i fits x_i. In the third step of the above algorithm, if association s_d fits more data records than any other associations, then we say s_d is the dominant association. We can use s_d as the granular descriptor of this cluster and use the coverage which is calculated from step 4 in the above algorithm as the effectiveness factor of this descriptor. In section 6.4 and section 6.5, we will use sets to describe the SOM clusters for synthetic data and software measure data.

6.3 Using Fuzzy Sets as Granular Descriptor of SOM

A fuzzy set is a superset of a set. In case of sets, an object is either completely inside a set or completely outside a set. There is a sharp boundary to define the membership of a set. In case of a fuzzy set, an object can have partial membership. The boundary to define membership does not have to be sharp. According to Formula 6.1, the membership degree in a set is either 0 or 1. In a fuzzy set, the membership degree is between 0 and 1. The formal definition of fuzzy set is: A fuzzy set is characterized by a membership function mapping the elements of a domain, space, or universe of discourse **X** to the unit interval [0, 1](Zadeh 1965).

Using sets, we can describe a SOM cluster as "LOC is *low* and NOM is *middle*", but we do not know the degree to which this cluster is compatible with this descriptor. Some clusters may be close to the boundary of the set and others may be in the center of the set, but they will all be described by the same descriptor. Using fuzzy set, we can not only describe a cluster by a suitable fuzzy set descriptor, but also provide the degrees to which this cluster is compatible with this descriptor. A fuzzy set descriptor of a software module cluster may look like "LOC is *low* with membership degree of 0.95 and NOM is *middle* with membership degree of 0.6".

In order to apply fuzzy sets, the first step is to construct a fuzzy set and a fuzzy membership function for each variable in a software measure data set. There are many shapes of membership function. The most frequently used one is triangular membership function and we will use it as fuzzy membership function to describe software modules. Because software measure data is usually skewed to the low end, we will use similar technique as discussed in section 6.2 to

construct fuzzy sets(Pedrycz, 2001c). Figure 6.4 illustrates the construction of fuzzy sets for one variable in a software measure data set.

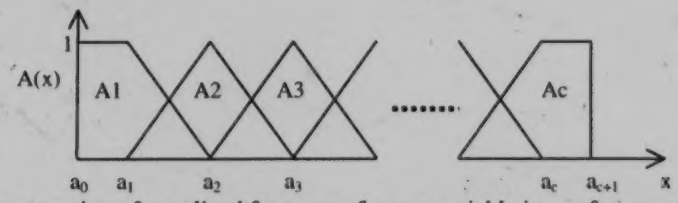


Figure 6.4 Construction of equalized fuzzy sets for one variable in a software measure data set There are c fuzzy sets, corresponding to A1, A2, ... Ac.

In Figure 6.4, a_0 is the minimum value of this software measure variable and a_{c+1} is the maximum value of this software measure variable. a_1, a_2, ... a_c are chosen to satisfy the following relation:

$$\frac{1}{N}\sum_{i=1}^{N'} A(x_i) = \frac{1}{2c} \tag{6.2}$$

N is the total number of data records in the data set
A(x) is the fuzzy membership function
x_i is the value of a data variable, $a_{k-1} \le x_i \le a_k$, k=1, 2... c
N' is the number of data records for which
this variable assumes values between a_{k-1} and a_k

The above construction of fuzzy sets ensures that all the software modules are sufficiently described. Otherwise, if we construct the fuzzy sets evenly over the range of software measure values which is similar to the construction in Figure 6.2, then most data records will only be described by one or two fuzzy sets and other fuzzy sets will only be used to describe few data records.

Once the fuzzy sets are constructed, we can proceed to describe clusters in SOM. In SOM, we can identify clusters using the clustering map and the data distribution map. For each cluster, we can list all the data records that are located in this cluster. Then we use the following algorithm to find fuzzy set descriptors for this cluster.

1. Combine all the fuzzy sets of all the variables to make c^n associations where c denotes the number of fuzzy sets and n denotes the number of variables in the software measure data set.
2. For each fuzzy set association s_k(k =1, 2... c^n), do the following
   - 2.1 For each data record x_i(i = 1, 2... N_{cl}, N_{cl} is the number of data records in this cluster), compute the t-norm result t_i of the membership degrees for all the fuzzy sets in association s_k.
   - 2.2 Compute the mean value m_k over all t_i(i = 1, 2... N_{cl})
3. Find the maximum value of m_k(k =1, 2... c^n) as m_d and pick the corresponding association s_d as the dominant fuzzy set association to describe this cluster.

From step 1 to 3, we get the dominant fuzzy set association s_d as the descriptor for this cluster. From step 4 to 5, we can find out the effectiveness factor of this descriptor to describe this cluster.

4. For each data record x_i(i = 1, 2... N_{cl}), compute the t-norm result t_i of the membership degrees for all the fuzzy sets in association s_d.

5. Take the average of all $t_i (i = 1, 2 \ldots N_{cl})$ as an effectiveness factor of the fuzzy set association s_d to describe this cluster.

In the above algorithm, the t-norm is realized in the form of the product(Pedrycz, 1998). Using the above algorithm, we can get a SOM cluster descriptor like "LOC is *low* with degree 0.9 and NOM is *middle* with degree 0.6". Obviously, fuzzy sets give us a more precise linguistic descriptor for the SOM clusters than sets do. Unlike numerical prototypes, fuzzy sets descriptors are easy to understand and we do not have to interpret the actual software measure values. In the next two sections, we will show some applications of fuzzy sets to describe SOM clusters.

6.4 Experiment with Synthetic Data

We use a 2-D synthetic data set to demonstrate the granular descriptors of SOM clusters. In a 2-D synthetic data set, we can plot and view what the actual clusters look like, so that we can evaluate the correctness and effectiveness of granular descriptors. Figure 6.5 shows the 2-D synthetic data set.

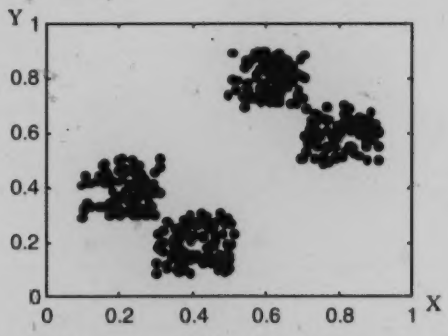


Figure 6.5 2-D synthetic data set.
There are 4 clusters in this data set.
Each cluster has 100 data points with Gaussian distribution

6.4.1 SOM for Synthetic Data

We train a SOM for this synthetic data set using the following parameters:

- Size of the maps: 10 x 10
- Normalization of variables: linear
- Initial neighborhood radius: 3
- Initial condition: randomly initiated connections
- No frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 3000 iterations

After training, we get a collection of maps that are shown in Figure 6.6. From the clustering map, we can easily determine the existence of 4 distinct clusters corresponding to the 4 clusters in the synthetic data set.

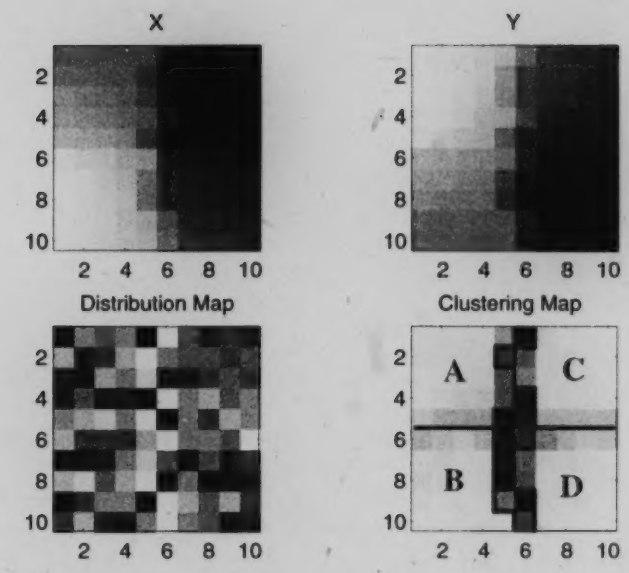


Figure 6.6 Self organizing maps for a 2-D synthetic data set
There are 4 clusters in the clustering map

6.4.2 Set Descriptors of SOM clusters for Synthetic Data

In order to apply sets to describe the SOM clusters, first we need to construct the sets. We use the technique discussed in Figure 6.3 to construct sets. Figure 6.7 shows the sets construction for the synthetic data set. In Figure 6.7, there are 3 sets constructed, so A1 denotes *low*, A2 denotes *middle*, A3 denotes *high*. If there are 5 sets constructed, then A1 denotes *very low*, A2 denotes *low*, A3 denotes *middle*, A4 denotes *high*, A5 denotes *very high*. We use these notations in all the experiments in this chapter.

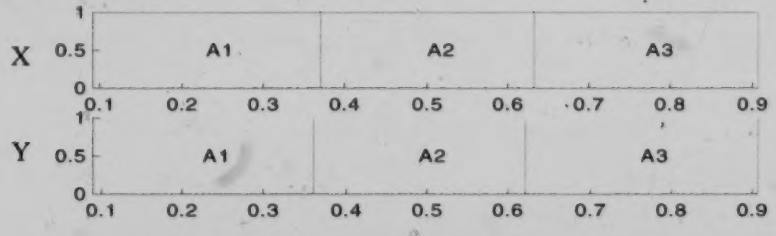


Figure 6.7 Equalized sets construction for the synthetic data
There are 3 sets constructed over the ranges of X and Y

Using the algorithm discussed in section 6.3, we get the following set descriptors for each cluster in Figure 6.6. In Table 6.1, "Data%" indicates the percentage of data in this cluster to the total data. "Coverage" indicates the percentage of data in this cluster that can be described by this sets association. We use these notations in all the following experiments. Here in Table 6.1, each cluster has approximately 1/4 of the total data. Each cluster has distinct descriptors. The descriptors can cover the majority of data in a cluster. Figure 6.8 visualizes the set descriptors for the synthetic data. In Figure 6.8, two set descriptors form a rectangular area. If the majority of data elements in a cluster are inside a rectangular area, then this cluster is effectively described by

the association of the two set descriptors. If we want to get higher coverage by set descriptors, we can apply the union of sets. For example, Cluster B can be described as "X is A1 and Y is A2 or A1", so that the set descriptor will have 100% coverage of data in cluster B.

Table 6.1 Set descriptors of SOM clusters for synthetic data using 3 sets

Cluster	Data%	Coverage	X	Y
A	23	0.85	A2	A1
B	25	0.81	A1	A2
C	23	0.79	A3	A2
D	23	0.84	A2	A3

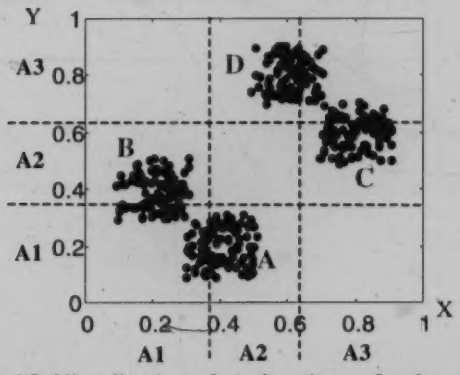


Figure 6.8 Visualization of set descriptors for the synthetic data

We can use more sets to describe the same clusters so that we can get finer granular descriptors. In Figure 6.9, we construct 5 sets for each variable in the synthetic data set. In Table 6.2, we show the cluster descriptors using 5 sets. According to Table 6.2, each cluster has distinct descriptors. Because we use more sets, the cluster descriptors are more detailed. However, the coverage of descriptor for each cluster is lower compared to Table 6.1. This is quite intuitive, because broader descriptors(fewer sets) tend to cover more content, while more specific descriptors(more sets) tends to cover less content.

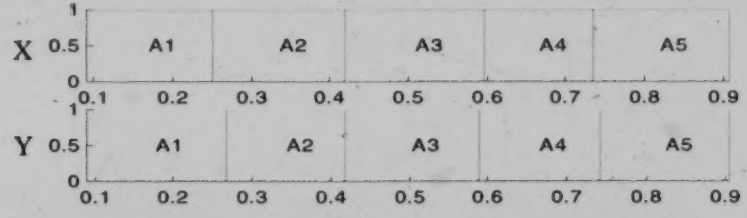


Figure 6.9 Equalized sets construction for the synthetic data set
There are 5 sets constructed over the ranges of X and Y

Table 6.2 Set descriptors of SOM clusters for synthetic data using 5 sets

Cluster	Data%	Coverage	X	Y
A	23	0.71	A2	A1
B	25	0.68	A1	A2
C	23	0.72	A5	A4
D	23	0.71	A4	A5

6.4.3 Fuzzy Set Descriptors of SOM clusters for Synthetic Data

In order to apply fuzzy sets to describe the SOM clusters, first we need to construct the fuzzy sets. We use the technique discussed in Figure 6.4 to construct fuzzy sets. Figure 6.10 shows the fuzzy sets construction for the synthetic data set.

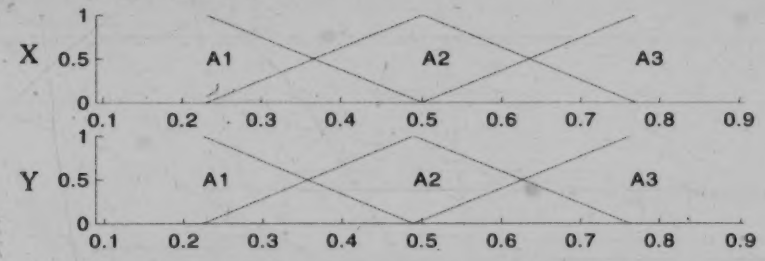


Figure 6.10 Equalized fuzzy sets construction for the synthetic data set
There are 3 fuzzy sets constructed over the ranges of X and Y

Using the algorithm discussed in section 6.3, we get the following fuzzy set descriptors for each cluster in Figure 6.6. In Table 6.3, "avg_D" indicates the average fuzzy membership degree of all the variables. "min_D" indicates the minimum fuzzy membership degree of all the variables. These notations is used throughout this chapter. We can see that the fuzzy set descriptors use the same "word" to describe cluster as sets. For example, both sets and fuzzy sets use "X is A2 and Y is A1" to describe Cluster A. However, in case of fuzzy sets, we also know the degree to which a cluster is compatible with its descriptors. For example, we can describe Cluster C as "X is A3 with degree 1 and Y is A2 with degree 0.75". In Table 6.3, the 4 clusters are effectively described by fuzzy sets.

Table 6.3 Fuzzy set descriptors of SOM clusters for synthetic data using 3 fuzzy sets

Cluster	Data%	avg_D	min_D	X	Y
A	23	0.9	0.81	A2(0.81)	A1(0.99)
B	25	0.89	0.78	A1(1)	A2(0.78)
C	23	0.87	0.75	A3(1)	A2(0.75)
D	23	0.88	0.76	A2(0.76)	A3(1)

We can use more sets to describe the same clusters so that we can get finer granular descriptors. In Figure 6.11, we construct 5 sets for each variable in the synthetic data set. In Table 6.4, we show the cluster descriptors using 5 fuzzy sets.

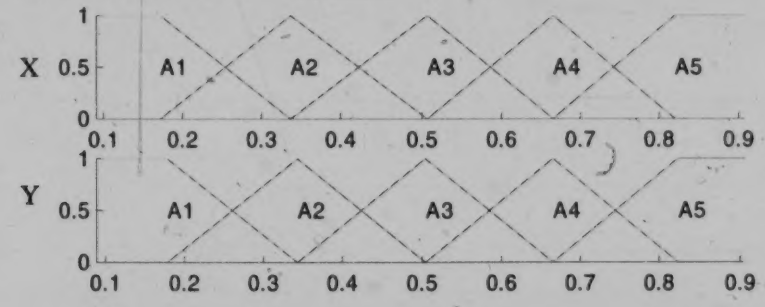


Figure 6.11 Equalized fuzzy sets construction for the synthetic data set
There are 5 fuzzy sets constructed over the ranges of X and Y

Table 6.4 Fuzzy Set descriptors of SOM clusters for synthetic data using 5 fuzzy sets

Cluster	Data%	avg_D	min_D	X	Y
A	23	0.82	0.73	A2(0.73)	A1(0.92)
B	25	0.86	0.79	A1(0.93)	A2(0.79)
C	23	0.86	0.8	A5(0.93)	A4(0.8)
D	23	0.88	0.8	A4(0.8)	A5(0.97)

In table 6.4, each cluster has distinct descriptors. All the clusters are described effectively. Compared with the 3-fuzzy-set descriptors, the 5-fuzzy-set descriptors have lower average degrees of membership. This is quite intuitive because more specific descriptors tend to cover less content.

6.4.4 Using Fuzzy Set Union to Describe Larger Clusters in Synthetic Data

We have seen the effectiveness of cluster descriptors using fuzzy sets in the above sections. Now let's see the use of the union of fuzzy sets to describe larger clusters. In Figure 6.6, Cluster A and Cluster B are similar, their boundary is not very distinct. Cluster C and Cluster D are also similar, their boundary is not very distinct. To generalize similar clusters to make a larger cluster, we merge Cluster A and Cluster B into one cluster, and merge Cluster C and Cluster D into one cluster. Then we use the union of fuzzy sets to describe these merged clusters. In Table 6.5, Cluster (A+B) covers almost half of the total data and so does Cluster (C+D). The membership degrees are higher than before the merge. We can describe Cluster (A+B) as "X is A1 or A2 with degree 0.93 and Y is A1 or A2 with degree 0.92".

Table 6.5 Using Fuzzy Set Unions to describe SOM clusters for synthetic data

Cluster	Data%	avg_D	min_D	X	Y
A+B	48	0.925	0.92	A1∪A2(0.93)	A1∪A2(0.92)
C+D	46	0.95	0.93	A4∪A5(0.93)	A4∪A5(0.97)

6.5 Experiment with Software Measure Data

The experiments on the 2-D synthetic data demonstrate very well how the granular descriptors can be used on SOM clusters. Now, we apply the granular descriptors to a real software measure data set, the Linguist, which has been introduced in section 4.2.2.

6.5.1 SOM for Software Measure Data

We train a SOM for this Linguist software measure data set using the following parameters:

- Size of the maps: 10 x 10
- Normalization of variables: logistic
- Initial neighborhood radius: 3
- Initial condition: randomly initiated connections
- No frequency sensitive learning
- Type of neighborhood: Gaussian Function
- Termination criterion: 3000 iterations

After training, we get a collection of maps which are shown in Figure 6.12. From the clustering map, we can select 5 clusters named as A, B, C, D and E.

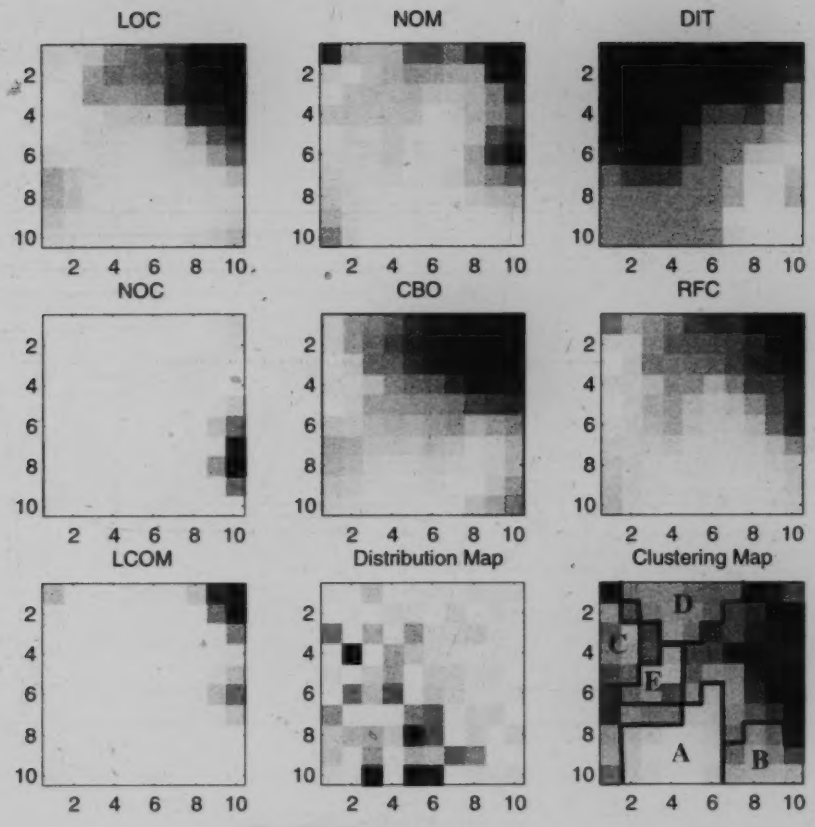


Figure 6.12 Self organizing maps for the Linguist software measure data set

6.5.2 Set Descriptors of SOM Clusters for Software Measure Data

In order to apply sets to describe the SOM clusters, first we need to construct the sets. We use the technique discussed in Figure 6.3 to construct sets. Figure 6.13 shows the sets construction for the software measure data. We can see that the sets here are quite different from the sets constructed for the 2-D synthetic data. In the set construction using the 2-D synthetic data, we use linear coordinates to plot sets. Here in Figure 6.13a, we use logarithmic coordinates to plot the sets for all software measures except DIT. This is because this Linguist data set has skewed histograms(see Figure 4.8) in all the measures except DIT. For those software measure variables that have skewed histograms, if we do not apply logarithmic coordinates, we will get a plot as shown in Figure 6.13b, for which it is difficult to identify low-value sets.

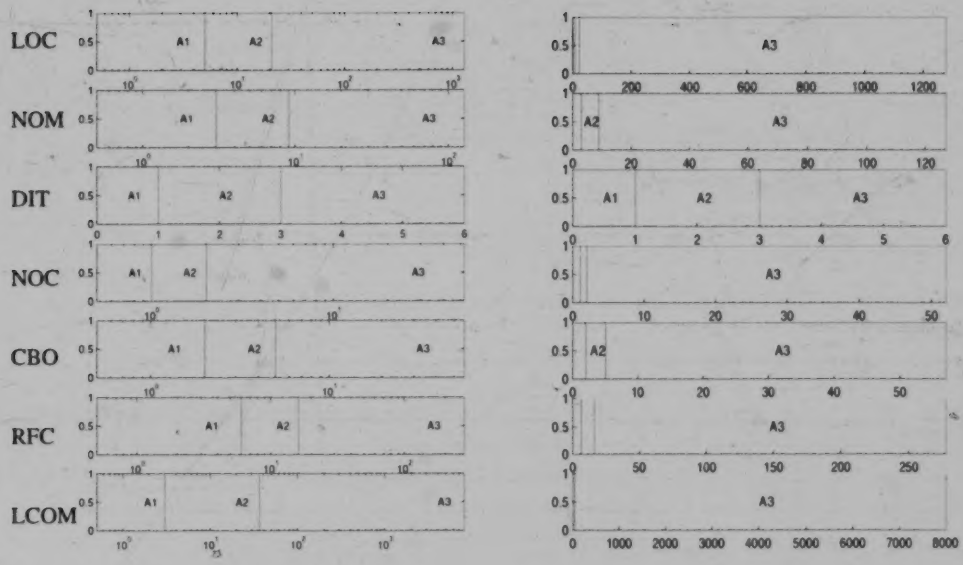


Figure 6.13 Equalized sets constructions for the Linguist software measure data set
There are 3 sets constructed over the ranges of each variable
(a) apples logarithmic coordinates; (b) applies linear coordinates

Using the algorithm discussed in section 6.3, we get the following set descriptors for each cluster in Figure 6.12. The five clusters represent 77.8% of total of data records. Each cluster is effectively described by sets. From the sets descriptors, we can easily determine the characteristics of each cluster. For example, Cluster A consists of software modules that have *low* values in all software measures except NOC, which has *middle* values. Cluster D consists of software modules that have *high* values in all software measures.

Table 6.6 Set descriptors of SOM clusters for the Linguist software measure data using 3 sets

Cluster	Data%	Coverage	LOC	NOM	DIT	NOC	CBO	RFC	LCOM
A	39	0.59	A1	A1	A1	A2	A1	A1	A1
B	9	0.78	A1	A1	A1	A2	A1	A1	A1
C	9.6	0.84	A1	A2	A3	A1	A1	A2	A2
D	8.2	0.6	A3	A3	A3	A3	A3	A3	A3
E	12	0.7	A2	A2	A2	A1	A2	A2	A2

We can use more sets to describe the same clusters so that we can get finer granular descriptors. In Figure 6.14, we construct 5 sets for each variable in the Linguist software measure data set. In Table 6.7, we show the cluster descriptors using 5 sets. In Table 6.7 we can see that each cluster has distinct descriptors, which clearly show the characteristics of the cluster. For example, in Cluster B, software modules have *low* values in all software measures except NOC, which has *high* values in this cluster. In Cluster D, software modules have *high* values in all software measures except NOC, which has *low* values in this cluster. Because we use more sets, the 5-set cluster descriptors are more detailed than the 3-set descriptors. However, the coverage of 5-set

descriptors for each cluster goes down compared to the 3-set descriptors in Table 6.6. This is quite intuitive, because broader descriptors(fewer sets) tend to cover more software modules, while more specific descriptors(more sets) tend to cover fewer software modules.

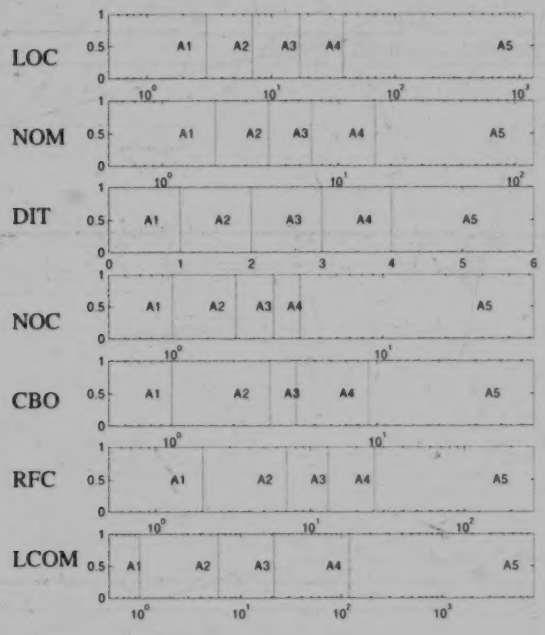


Figure 6.14 Equalized sets construction for the Linguist software measure data set
There are 5 sets constructed over the ranges of each variable

Table 6.7 Set descriptors of SOM clusters for the Linguist software measure data using 5 sets

Cluster	Data%	Coverage	LOC	NOM	DIT	NOC	CBO	RFC	LCOM
A	39	0.44	A2	A1	A2	A3	A1	A1	A1
B	9	0.7	A1	A1	A1	A4	A1	A1	A1
C	9.6	0.74	A1	A3	A5	A1	A2	A2	A5
D	8.2	0.51	A4	A4	A4	A1	A4	A4	A4
E	12	0.62	A3	A3	A4	A2	A3	A4	A3

6.5.3 Fuzzy Set Descriptors of SOM clusters for Software Measure Data

In order to apply fuzzy sets to describe the SOM clusters, we first need to construct them. We use the technique discussed in Figure 6.4 to construct fuzzy sets. Figure 6.15 shows the fuzzy sets construction for the Linguist software measure data set. Using the algorithm discussed in section 6.3, we get the fuzzy set descriptors listed in Table 6.8. Compared with Table 6.6, the fuzzy set descriptors use the same word as the set descriptors to describe clusters. Using fuzzy set descriptors, we can also know the degree to which the cluster is compatible with the descriptor. For example, in Cluster A, RFC is A1(*low*) with membership degree of 0.97, which means that RFC is undoubtedly *low*. One the other hand, NOC is A2(*middle*) with membership degree of 0.51, which means that NOC is halfway between *low* and *middle*.

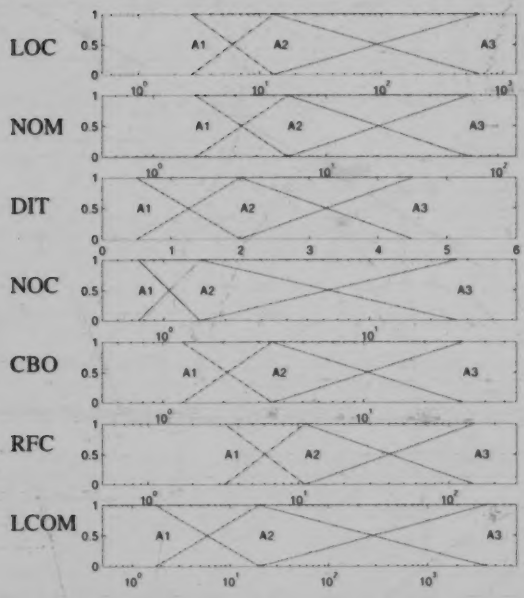


Figure 6.15 Equalized fuzzy sets construction for the Linguist software measure data set
There are 3 fuzzy sets constructed over the ranges of each variable

Table 6.8 Fuzzy Set descriptors of SOM clusters
for the Linguist software measure data using 3 fuzzy sets

Cluster	Data%	avg_D	min_D	LOC	NOM	DIT	NOC	CBO	RFC	LCOM
A	39	0.83	0.51	A1 (0.88)	A1 (0.89)	A1 (0.72)	A2 (0.51)	A1 (0.89)	A1 (0.97)	A1 (0.96)
B	9	0.8	0.69	A1 (0.74)	A1 (0.79)	A1 (0.99)	A2 (0.69)	A1 (0.71)	A1 (0.86)	A1 (0.78)
C	9.6	0.87	0.61	A1 (0.89)	A2 (0.87)	A3 (0.96)	A1 (1)	A1 (0.88)	A2 (0.61)	A2 (0.9)
D	8.2	0.78	0.57	A3 (0.67)	A3 (0.73)	A3 (0.96)	A3 (0.73)	A3 (0.77)	A3 (0.73)	A3 (0.89)
E	12	0.88	0.69	A2 (0.84)	A2 (0.92)	A2 (0.69)	A1 (1)	A2 (0.88)	A2 (0.9)	A2 (0.91)

We can use more fuzzy sets to describe the same clusters so that we can get more detailed granular descriptors. In Figure 6.16, we construct 5 fuzzy sets for each variable in the Linguist software measure data set. In Table 6.9, we show the cluster descriptors using 5 fuzzy sets. Compared with the 3-fuzzy-set descriptors, the 5-fuzzy-set descriptors have lower average degree of membership. This is quite intuitive since more specific descriptors tend to cover fewer software modules.

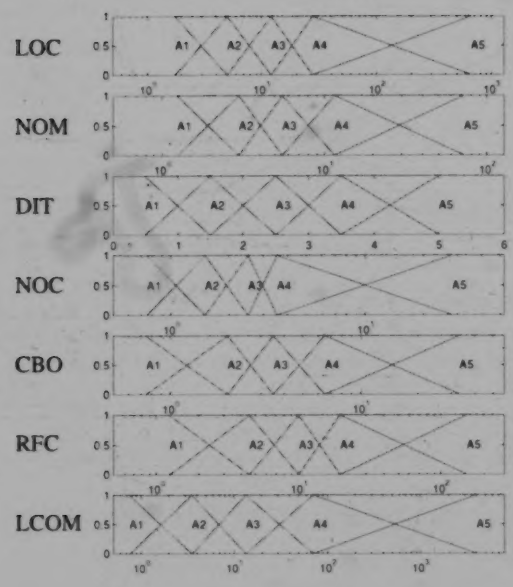


Figure 6.16 Equalized fuzzy sets construction for the Linguist software measure data set
There are 5 fuzzy sets constructed over the ranges of each variable

Table 6.9 Fuzzy Set descriptors of SOM clusters
for the Linguist software measure data using 5 fuzzy sets

Cluster	Data%	avg_D	min_D	LOC	NOM	DIT	NOC	CBO	RFC	LCOM
A	39	0.71	0.41	A2 (0.8)	A1 (0.71)	A2 (0.82)	A3 (0.41)	A1 (0.65)	A1 (0.74)	A1 (0.84)
B	9	0.69	0.55	A1 (0.55)	A1 (0.75)	A1 (0.96)	A4 (0.57)	A1 (0.61)	A1 (0.7)	A1 (0.68)
C	9.6	0.76	0.59	A1 (0.7)	A3 (0.73)	A5 (0.91)	A1 (1)	A2 (0.66)	A2 (0.72)	A5 (0.59)
D	8.2	0.74	0.55	A4 (0.78)	A4 (0.74)	A4 (0.55)	A1 (0.8)	A4 (0.69)	A4 (0.8)	A4 (0.78)
E	12	0.78	0.57	A3 (0.8)	A3 (0.77)	A4 (0.98)	A2 (0.57)	A3 (0.79)	A4 (0.67)	A3 (0.88)

6.5.4 Using Fuzzy Set Union to Describe Larger Clusters in Software Measure Data

Similar to the 2-D synthetic data set, we can also use union of fuzzy sets to describe larger clusters in this Linguist software measure data set. We merge Cluster A and Cluster B to form a larger cluster (A+B). We merge Cluster C, Cluster D and Cluster E to form a larger cluster (C+D+E). Then we use the fuzzy set union to describe the two big clusters. According to Table 6.10, Cluster (A+B) has 48% of total software modules. These modules are basically small because most measures are described by A1(*very low*) and A2(*low*). Cluster (C+D+E) has 29.8% of the total software modules. The software modules in Cluster (C+D+E) are more heterogeneous than the software modules in Cluster (A+B), because they need more fuzzy sets to make union to describe them. For example, the LCOM values for the software modules in Cluster (A+B) can be

described as A1(*very low*), but the LCOM values for the software modules in Cluster (C+D+E) have to be described as A3(*middle*) or A4(*high*) or A5(*very high*).

Table 6.10 Using Fuzzy Set Unions to describe SOM clusters for the Linguist software measure data

Cluster	Data%	avg_D	LOC	NOM	DIT	NOC	CBO	RFC	LCOM
A+B	48	0.76	A2∪A1 (0.8)	A1 (0.75)	A2∪A1 (0.96)	A3∪A4 (0.57)	A1 (0.65)	A1 (0.74)	A1 (0.84)
C+D+E	29.8	0.86	A1∪A3∪ A4 (0.8)	A3∪A4 (0.77)	A4∪A5 (0.98)	A1∪A2 (1)	A2∪A3 ∪A4 (0.79)	A2∪A4 (0.8)	A3∪A4 ∪A5 (0.88)

6.6 Summary

In this chapter, we introduced the granular description of SOM clusters. We used sets and fuzzy sets to describe the characteristics of SOM clusters. When using SOM to analyze software measure data, once we identify the clusters, we can apply granular descriptors to describe these clusters. Compared with the numerical descriptors(prototypes), the granular descriptors are easier to understand. The granular descriptors provide with us with an objective way to describe SOM clusters using words.

7 SOM Tool Design

In order to apply SOM to analyze software measures, we need to adjust many parameters related to SOM training. Once the SOM is trained, we need to interact with different maps such as by selecting interesting pixels and performing further analyzing the data in a certain cluster. All of these adjustments and user interactions require a tool with a friendly user interface. In the following sections, we will now discuss the design of a SOM tool for software measure analysis.

7.1 Requirement Analysis

Currently, there are some SOM tools in Matlab Toolbox. These tools all have some obvious disadvantages:

- Must be run within Matlab which is a large software.
- Do not have a convenient user interface to analyze SOM clusters.
- Do not have enough result management capability.
- Do not have any analysis routines specific to software measures.

In order to eliminate the shortcomings of these Matlab tools, our SOM tool will have the following features:

1. Stand-alone and compact
   Only one executable file is required for running the SOM Tool program. The size of the executable file is small so that it is easy to download it from Internet. The executable file can be run directly through an operating system, rather than requiring supporting software such as Matlab.

2. Easy adjustment of parameters
   All the parameters of SOM algorithm can be adjusted. These parameters include:
   - Map size
   - Neuron shape
   - Initial neighborhood radius
   - Learning rate
   - Learning epochs
   - Neighborhood functions
   - Learning algorithms
   - SOM initialization method
   - Data normalization method

3. Interactive analysis of result
   The most attractive feature of SOM is its interactive visualization. After the SOM is trained, we can visualize the weight maps, the clustering map and the data distribution map. In addition to displaying maps, this tool should also allow the user to interactively explore the maps. The user should be able to mark some interesting neurons and check what data is attached to these neurons. The user should also be able to select a cluster and train a new SOM based on the corresponding data, and finally generate a hierarchical SOM(See detailed discussion in section 4.4.2).

4. Data, parameter and result management
   After training on a data set, a SOM weight matrix is generated. When saving this weight matrix into file, we need to save all the training parameters. With these training parameters

linked to the weight matrix, we can compare the results for different SOM parameters and continue training from a saved training session.

7.2 *User interaction modeling*

According to the requirement analysis, we can list the use cases(Boggs, 1999) for the SOM Tool:

1. Load data file, select or remove some measures for analysis
2. Set training parameters.
3. Set the display preference for visualization.
4. Start, suspend or stop training.
5. Show training process dynamically.
6. Save learned SOM and corresponding parameters.
7. Load existing SOM and corresponding parameters.
8. Display maps(the weight maps, distribution map and the clustering map).
9. Show statistics for the training data set(Histograms, correlation matrix)
10. Pick interesting neurons and do cluster analysis(List data, grow child SOM)
11. Map testing data to the trained SOM and see how they are located.

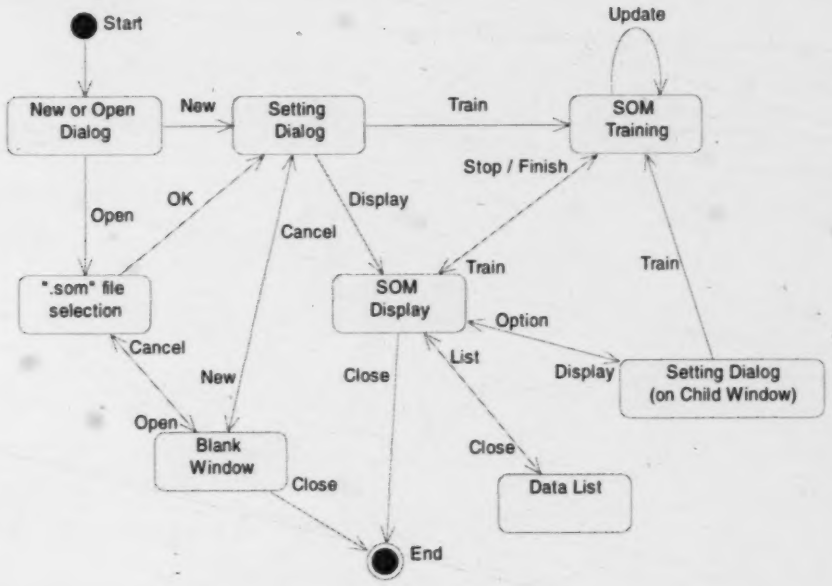


Figure 7.1 State diagram for user interaction modeling

This SOM Tool should have the functionality to satisfy the above use cases. The user interface should also be designed to reflect how the user does experiments in the real world.

Figure 7.1 shows the state diagram to model the user interaction with the SOM Tool. Basically, there are 10 states in this tool. Each state is either a dialog box or a window. Depending on how the user operates, different states can be entered. Some states can only appear once, such as "Start", "New or Open Dialog" and "End". Others can be entered more than once.

7.3 *User interface architecture design*

According to the functions discussed in the previous section, the basic user interface architecture can be illustrated by the following figures.

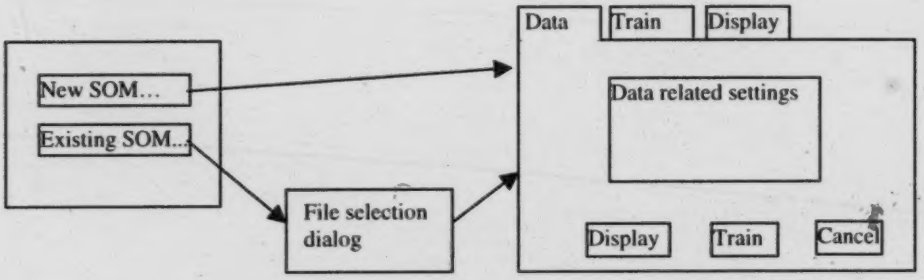


Figure 7.2 Simplified screen flow chart for the SOM Tool

When the user starts to use the tool, he or she has two options: train a new SOM or open an existing SOM. Both options will lead to a "setting" screen as shown in Figure 7.2. In the "setting" screen, the user can set different parameters that are organized into three folders: Data Setting, Training Setting and Display Setting. In the Data Setting folder, the user can browse the data file name(if training a new SOM), add or remove some measures from the data set, and add comments to the data set. In the Training Setting folder, the user can set the size and shape of SOM, choose different parameters, and change the default training parameters. In the Display Setting folder, the user can determine the color, size and layout of the maps and histograms. Next, the user can start to train the SOM, or simply display it. If the user chooses "Cancel", all the changes is ignored and nothing is displayed in the client area.

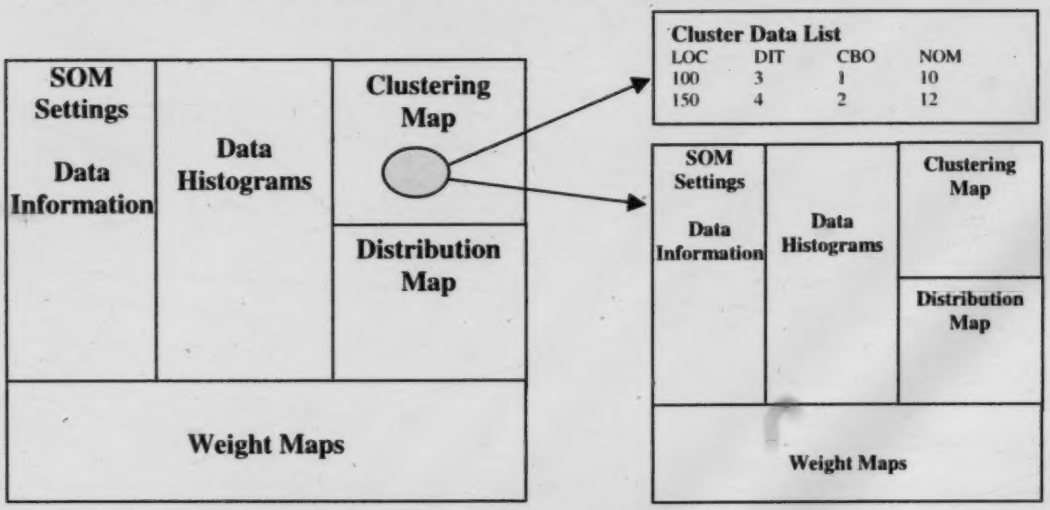


Figure 7.3 Screen partition for result display and cluster analysis

Figure 7.3 shows how to display the SOM results. The "result" screen is divided into five panes. The upper-left pane displays the SOM setting, the dynamic training process, the data file information and the correlation matrix for the training data set. The data histogram pane shows

the histograms for all the selected measures. The clustering map pane displays the clustering map for the training data set. The distribution map pane displays the data distribution map for the training data set. The Weight map pane shows the weight maps for all the selected measures. The size of each pane can be adjusted. In the clustering map pane and the distribution map pane, the user can mark interesting clusters for further analysis. The user can list the raw data records that are located in this cluster. The user can also train a new SOM based on the data in this cluster. This child SOM growing process can be continued until there are not enough data in the cluster to train a SOM. From this SOM growing process, we can eventually generate a SOM hierarchy(Suganthan, 1998, 1999).

7.4 Major object identification

In order to apply OOD(Object Oriented Development) for the construction of this tool, we need to identify the major objects before implementation(Larman, 1997). An object should be an entity, clearly distinguishable from other objects. An object should encapsulate its own data and behavior, and should be as independent of other objects as possible. Some objects are specific to the generic user interface framework that is discussed in the next section. Following are some objects that are specific to this SOM Tool.

- Dataset
  This object encapsulates the data file information, measure descriptions, and the training data matrix. It also includes many operations to be applied to its encapsulated data. For example, it has a Normalize function to do data normalization and de-normalization.

- SOM
  This object encapsulates the SOM training parameters such as SOM size, training epochs and training rate. It also includes all the SOM algorithm computation, such as the function to train the SOM, the functions to make the clustering map and the data distribution map.

- Display
  This object encapsulates the display parameters such as color, shadow, edge, size and layout. It also handles the parameter validation.

- Map
  This object encapsulates the data necessary to draw a map. These data include map grid width and height, grid number per row and per column, location of map, map title and legend. It also encapsulates the operations to draw map grid, draw whole map, and draw title and legend.

- Histogram
  This object encapsulates the data necessary to draw a histogram. These data include the number of bins for the histogram, the count for each bin, the axis scales for the histogram and the title of the histogram. It also encapsulates the operations to draw bins, axes, labels and titles.

7.5 Implementation

All the above discussions are platform and language independent. We can apply the above architectural design to any platforms using any programming languages. In this section, we discuss what platform and programming language we use.

7.5.1 Platform

The two most frequently used platform are Microsoft(MS) Windows family and the Unix family. MS Windows has a more friendly graphical user interface and is more popular for end users. The Unix family is more robust for large business servers but has a weaker graphical user interface. Because this SOM tool is going to be a graphics intensive program and is used mostly by ordinary users, we choose MS Windows NT as its platform.

7.5.2 Programming Language

To develop a friendly user interface, there are some languages from which to choose: Visual Basic(VB), Java and C++. They all support Graphical User Interface(GUI) programming under the Windows platform. Visual Basic works as a component glue. It can integrate the forms, menus and dialog boxes and make development very fast. However, in this SOM tool development, there is intensive computation and the computation overheads will cause severe performance degradation. VB is a high level language and will require too many overheads, so it is not suitable for our SOM tool development. Java is good for network programming and is platform independent, but it has to run on a Java virtual machine which makes Java code slow. C++ is the most efficient language for Windows development. It supports pointers that are very flexible and efficient. Although C++ code is much more difficult to write than VB and Java, considering its efficiency, we still choose it as the programming language for this SOM tool.

Once we have decided to use C++, it is natural to choose Microsoft Visual C++ as the programming Integrated Development Environment(IDE), because it is the most popular IDE for C++ Windows development and has comprehensive documentation(Horton, 1998).

7.5.3 Framework

Using Visual C++, we can take advantage of Microsoft Foundation Class Library(MFC) which provides us with frameworks to build Windows applications. MFC can provide many skeletons(frameworks) for certain types of applications. For our SOM tool development, we employ Multiple Document Interfaces(MDI) framework.

MDI framework is provided by MFC(Richard, 1999). It has one main window and multiple child windows. Multiple child windows can be opened simultaneously, but only one of them can be active(has focus). Each child window can be re-sized, moved, but cannot be outside the main window. Each child window is associated with a data file. All the operations and results are done in this child window. Any two child windows are independent. For our SOM tool application, we use the MDI framework. Each training session is associated with one child window. We can open multiple data file and do multiple training sessions in different child windows without interference.

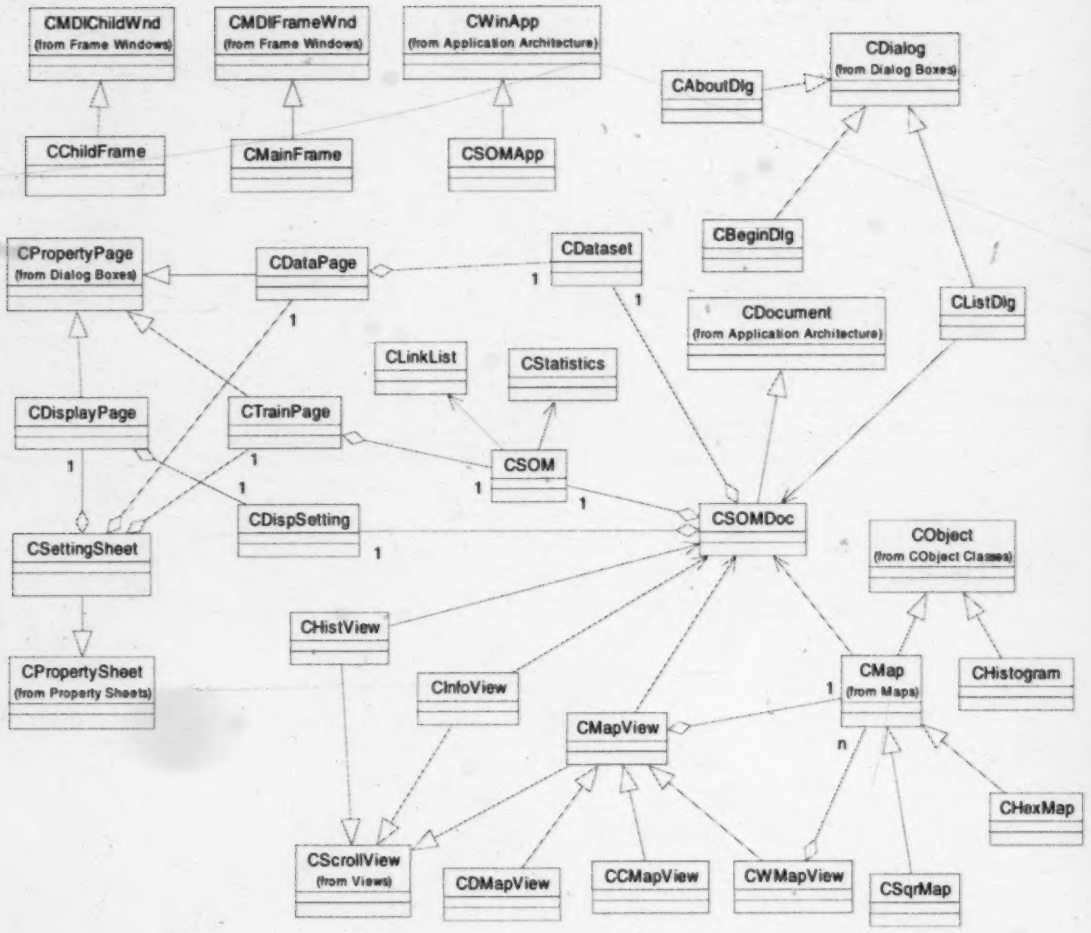
The MDI framework applies a very popular pattern: Document/View Pattern(Observer Pattern)(Larman, 1997). Document includes all the data information such as the parameters and the data file. Generally, one document is associated with one data file. It handles the disk operation and data storage. View is the presentation of the document. A document can have more than one presentation. For example, we have a document that stores an array of integers. We may use one view to show the text list of this array. At the same time, we may use another view to show a curve for this array. Both text list and curve are based on the data in the document. Any change in the document must be reflected in all the views. In our SOM tool application, the training, display parameters and the training data information for one training session are saved in one document. Multiple documents can exist in the main window simultaneously. Each document or training session is associated with five views. Each view shows a specific aspect of the

document, including the clustering map, the data distribution map, data histograms, the weight maps and text information. They correspond to the five panes shown in Figure 7.3. If the document changes, it must notify all its associated views so that they can be updated. On the other hand, if we modify one view, we must do the same modification to its associated document, then the document will again update all other views.

Unlike the standard MDI framework which associates each child window with one view(Richard, 1999), in this SOM tool, we associate one document with one child window and associate this child window with multiple views. This is achieved by dividing the child window into many panes(see Figure 7.3). Each pane is associated with one view and is responsible for displaying one aspect of the document. In our SOM tool, the main window manages all the documents, child windows and menus. The document manages the data. The child window manages its five panes(views). The views(panes) manage the display and user interaction work.

7.5.4 Overall Class Diagram

Figure 7.4 shows the class diagram for this SOM tool. CSOMApp manages the main program process. CMainFrame and CChildFrame manage the main window and the child windows. CBeginDlg corresponds to the startup dialog box. CListDlg corresponds to the data list dialog box. It manages the list of total training data and the cluster data. CSettingSheet corresponds to the setting dialog box. It contains 3 setting pages: CDataPage, CTrainPage and CDisplayPage. These setting pages manage the training data setting, the SOM setting and the display setting. CSOM is the class to implement SOM object which is discussed in the previous section. CDataSet and CDispSetting implement the Dataset object and Display object, respectively. CMap implements the Map object. It has two children: CSqrMap and CHexMap, which correspond to the square shape neuron and the hexagon shape neuron, respectively. CMapView manages the display and mouse actions. It has 3 children: CDMapview, CCMapView and CWMapView, which correspond to the data distribution map, the clustering map and the weight maps, respectively. CHistView and CInfoView manage the display of histograms and text information, respectively. CSOMDoc is the most important class in this diagram. It manages all the data corresponding to one training session. It contains one instance of CDispSetting, one instance of CSOM and one instance of CDataset. All the classes that use data will talk to this class. Any modification of data that is saved in this CSOMDoc must be reflected in other data consumer classes.



Navigation Relationship
Class A can navigate to class B, but class B can not navigate to class A. In other words, class B is visible to class A but class A is invisible to class B.

Inheritance Relationship
Class A is inherited from class B. Class B is the parent class of class A.

Containment Relationship
Class B contains class A. Class A is a component of class B.

Figure 7.4 UML Class diagrams for SOM Tool implementation
This class diagram focuses on the relationship among classes, so all the member variables and member functions are omitted.

7.6 *Testing*

After this tool was implemented, we used various software project data sets to do the functional test. For example in section 4.4.1 and 4.4.2, the analysis was done using this SOM Tool. In this section, we demonstrate the performance of this tool.

7.6.1 Handling Large Data Set

To evaluate a data analysis tool, one critical factor is the size of data it can process. For this SOM Tool, we test the different behavior to process different size of data. First, we make 8 synthetic data sets. They have different number of variables, including 10, 20, 50 and 200. They also have different number of records, including 1000 and 10000. We used a typical SOM with 20 x 20 neurons to run these synthetic data sets. We used the default values for other SOM parameters supplied by the tool. Table 7.1 lists the behavior of the SOM Tool when handling different size of training data.

Table 7.1 Testing the ability of SOM Tool to handle large data set

#Variables / #Records	10	20	50	200
1000	Runs well, with fast response to user interaction during training.	Runs well, with fast response to user interaction during training.	Runs well, with slow response to user interaction during training.	Runs, but very sluggish to user interaction during training.
10000	Runs well, with fast response to user interaction during training.	Runs well, with slow response to user interaction during training.	Runs, but has no response to user interaction during training.	Program is frozen after loading data file.

According to the above test, this SOM Tool has a smooth degradation of performance when increasing the size of training data. When the training data set is small, such as 1000 x 10 (1000 records and 10 variables), the tool runs very well. Both the computation and the dynamic graphics run well. It has fast response to user's interactions, such as "Suspend" and "Stop" during training. When the training data set becomes large, the responses to user's interactions become slow, but the computation and dynamic graphics are still normal. When the data set size reaches 10000 x 50, the tool will have no response to user's interaction during training. When the data set size reaches 10000 x 200, the tool can load the data from a file, but once it begins to "Display" or "Train", the program is frozen. However, the program does not crash. It is just frozen and still has chance to revive after a long time. Therefore, we can conclude that this SOM Tool can handle large data sets with smooth degradation in performance. It does not crash suddenly when processing large data sets.

7.6.2 Computation Speed

In addition to performance on large data sets, the computation speed is also a very important index for evaluating a SOM tool. Since this tool is written in C++, it should be much faster than the Matlab code using same algorithm. We used a machine with 450M Pentium III CPU and 64M memory running Windows NT 4.0. Table 7.2 shows a comparison of computation time between the SOM Tool and the Matlab code. We use 6 synthetic data sets, with linearly increasing sizes. The SOM parameters for this tool and the Matlab code are identical. The underlying algorithms are also identical. According to Table 7.2, the C++ code is much faster than the Matlab code. This SOM Tool can reduce the training time by a factor of about 11 compared with its Matlab counterpart.

Table 7.2 A comparison of computation time between SOM Tool and Matlab code
The SOM size is 10 x 10. The learning epochs are 50 for rough training and 100 for fining tuning.
The time is measured by second.

Data set size	500 x 10	1000 x 10	1500 x 10	2000 x 10	2500 x 10	3000 x 10
SOM Tool	7	13	20	28	34	41
Matlab code	82	151	239	324	402	495

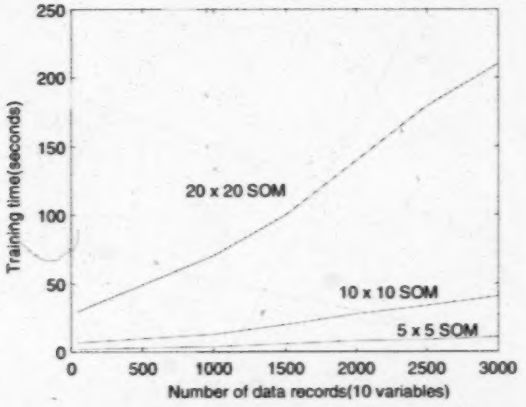


Figure 7.5 The training time for different size of data set using SOM Tool
The learning epochs are 50 for rough training and 100 for fining tuning

We also test the computation time of SOM Tool for different size of SOM. Figure 7.5 shows the relationship between the size of training data and the training time. In Figure 7.5, the data sets all have 10 variables. We can see that for the same SOM size, the training time is approximately linear to the size of training data. However, for the same training data set, the training time is not linear to the size of SOM. This implies that we must be careful to choose the size of SOM, because different size of SOM will need significantly different amount of time to train.

7.7 Future Improvements

So far we have successfully built a tool to demonstrate the ability of SOM for software measure analysis. Using this tool, we can select training data, set SOM parameters and train SOM. We can interactively analyze the training results. We can also manage the training results easily. However, the following features are still not available in this tool. They is implemented in the future version.

- More training algorithms
  Currently, we are using online training. We may add offline training in the future version. In addition, we could implement the fast winner searching algorithm that will greatly improve the training speed(Kaski, 1999a).

- Mark different clusters simultaneously
  The current version of SOM Tool provides the functions to mark map cells and analyze marked cluster. All the marked cells are considered as one cluster. Sometimes, users may want to mark several different clusters in the same map. This is not supported in the current version and is considered in the future version.

- Merge different clusters
  In the trained SOM, we can find different clusters. Sometimes, after checking the underlying data, the user may decide to merge these clusters into one cluster. The SOM can then be trained with this new rule. In the future version, we will consider this function.

- Export information
  Once we get the trained SOM, histograms and the correlation matrix. We may desire to do further analysis in Matlab. In the next version, the SOM Tool will have the capability to export all the training results in a format that can be read by Matlab.

7.8 Summary

In this chapter, we have systematically discussed the specification, design, implementation and testing of a SOM Tool for software measure analysis. Using this SOM Tool, users can interactively analyze the software measure data and easily find the software module clusters and feature associations. The development process discussed in this chapter also gives us an OOD(Object Oriented Development)example. We expect that it is helpful for the development of similar tools.

8 Conclusion

We have discussed a neural-computing approach for analyzing software measures. For software measures, which sometimes do not find themselves very well to statistical analysis, SOM(Self Organizing Maps) can give us an alternative way to analyze them. Using SOM, software measure relations and software module clusters can be easily visualized. This helps the software managers and developers to understand the internal properties of software modules and projects, and further to predict their external behaviors. The following is a recapitulation of the topics we discussed in this thesis.

- Software measurements
  Software measurements are the first step to apply quantitative software engineering in software management. The motivation, classification and characteristics of software measures were introduced. Some frequently used software measures were discussed. As a comparison to SOM analysis, the conventional statistical analysis of software measures was introduced.

- The self organizing maps
  An in-depth investigation of SOM algorithm is done here. First, the general SOM concept and basic SOM training algorithm were introduced. Second, the SOM parameters and their influence on the SOM results were discussed. Then, in order to make maximum use of SOM training results, we introduced some interpretation methods of SOM. Finally, some SOM improvements were discussed, such as data normalization and training speedup.

- Application of SOM to software measure analysis
  In this part, we first compared SOM with statistics in software measure analysis. Then three real software projects were used to demonstrate the application of SOM in software measure analysis, such as distribution, association and cluster analysis.

- Using different similarity measures in SOM
  The core of SOM algorithm is similarity comparison. The basic SOM applies Euclidean distance function to do the similarity comparison. However, this is not necessarily required in software engineering, where we need to compare the similarity between software modules. In this part, we first compared three frequently used distance functions in SOM. Then we proposed a neural network based similarity measure.

- Granular description of SOM clusters
  When using SOM to analyze software measure data, once we identify the clusters, we can apply granular descriptors(sets or fuzzy sets) to describe these clusters. Compared with the numerical descriptors(prototypes), the granular descriptors are easier to understand. The granular descriptors provide us with an objective way to describe SOM clusters with words.

- SOM tool design
  The SOM analysis needs intensive graphics and user interactions. Therefore, we designed a SOM tool, which provides users with a friendly environment to analyze software measures with SOM.

To summarize, SOM is a user-friendly and interactive visualization tool that helps software managers and developers get a thorough insight into the structure of the software module clusters and the related software measures. In other words, software managers and developers can get a global impression of the software project, module clustering and feature relationships by means

of a collection of maps. After that, if precise numbers are required, other analysis approaches such as statistics can be applied based on the results of the SOM analysis.

References

Anderson, T. M., (1984), *An Introduction to Multivariate Statistical Analysis*, John Wiley & Sons Canada, 1984

Bezdek, J. C., (1981), *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, N. York, 1981

Bezdek, R. and F. Ehlich., (1984), *FCM: The fuzzy C-means clustering algorithm*. Comp. Geoci., 10:191-023, 1984.

Bishop, Robert H., (1999), *Learning with LabVIEW*, Addison-Wesley Pub Co, Feb 1999.

Boggs, Wendy, Michael Boggs, (1999), *Mastering UML with Rational Rose*, Sybex, July 1999

Briand, L.C., J. Wüst, (1999), *The Impact of Design Properties on Development Cost in Object-Oriented Systems*, Technical Report ISERN-99-16

Chakraborty, Goutam and Basabi Chakraborty, (2000), *A Novel Normalization Technique for Unsupervised Learning in ANN*, IEEE Transactions on Neural Networks, VOL. 11, NO. 1, pages 253-257, 2000

Chidamber, Shyam R., and Chris F. Kemerer, (1994), *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, VOL 20, NO. 6, pages 476-486, June, 1994

Demarco, Tom, (1982), *Controlling Software Projects: Management, Measurement and Estimation*, Yourdon Press, New Jersey, 1982

Fenton, N. E., S. L. Pfleeger (1997), *Software Metrics: A Rigorous and Practical Approach*, PWS, London, 1997

Fenton, Norman E., (1999), *A Critique of Software Defect Prediction Models*, IEEE Transactions on Software Engineering, VOL 25. 5, pages 675-687, September/October 1999

Flanagan,. John A., (1996), *Self-organization in Kohonen's SOM. Neural Networks*; Neural Networks,VOL 9, NO.7, pages 1185-1197, 1996

Goppert, J. and W. Rosenstiel, (1995), *Neurons with continuous varying activation in self-organizing maps*. In J. Mira and F. Sandoval, editors, From Natural to Artificial Neural Computation. International Workshop on Artificial Neural Networks. Proceedings, pages 419-26. Springer-Verlag, Berlin, Germany, 1995

Horton, Ivor, (1998), *Beginning Visual C++ 6*, Wrox Press, 1998

Kaski,. Samuel, (1997), *Data exploration using self-organizing maps*. Acta Polytechnica Scandinavica, Mathematics, Computing and Management in Engineering Series No. 82, March 1997. DTech Thesis, Helsinki University of Technology, Finland.

Kaski, Samuel, Janne Nikkilä, and Teuvo Kohonen., (1998), *Methods for interpreting a self-organized map in data analysis*. In Michel Verleysen, editor, Proceedings of ESANN'98, 6th European Symposium on Artificial Neural Networks, Bruges, April 22-24, pages 185-190. D-Facto, Brussels, Belgium

Kaski, Samuel, (1999a), *Fast Winner Search for SOM-Based Monitoring and Retrieval of High-Dimensional Data*, In Proc. of ICANN'99, Edinburgh, UK, 7-10

Kaski, Samuel, Jarkko Venna, and Teuvo Kohonen, (1999b), *Coloring that Reveals High-Dimensional Structures in Data*, Proc. ICONIP'99

Kiviluoto, K. (1996), *Topology preservation in self-organizing maps*, In Proc. ICNN96, IEEEE Int. Conf. On Neural networks, 1996

Kohonen, Teuvo, (1995), *Self Organizing Maps*, New York : Springer, 1995

Kohonen, T., (1999), *Spotting Relevant Information in Extremely Large Document Collections*, 6th Fuzzy Days, Dortmund, Germany, May 25-28, 1999. Berlin 1999, Springer, pp. 59-61

Larman, Craig, (1997), *Applying UML & Patterns: An Approach to Object-Oriented Analysis & Design*, Prentice Hall Canada, 1997

Li, W., S. Henry (1993) *Object Oriented Metrics that Predict Maintainability*, Journal of Systems and Software, 23(2), 1993

MacDonell, Stephen G., (1994), *Statistical Analysis Procedures for Software Complexity Assessment Data*, New Zealand Journal of Computing 5(1):67-75, 1994

MacDonell, S.G. and A.R. Gray, (1997), *A Comparison of Modeling Techniques for Software Development Effort Prediction*, In Proc. of ICNIPIIS 1997, Dunedin, New Zealand, pages 869-872, 1997

McCabe, T. J., (1976), *A Complexity Measure*, IEEE Trans. Software Eng. 2, 308-320(1976)

Munson, John C., Taghi M. Khoshgoftaar, (1996), *Software Metrics for Reliability Assessment*, Handbook of Software Reliability Engineering, IEEE Computer Society Press and McGraw-Hill Book Company, 1996

Pedrycz, W., F. Gomide, (1998), *An Introduction to Fuzzy Sets*, MIT Press, 1998

Pedrycz, W., G. Succi, P. Musílek, X. Bai, (2001a), *Using Self-Organizing Map to Analyze Object-Oriented Software Measure*, Journal of Systems and Software, 2001, to appear

Pedrycz, W., G. Succi, M. Reformat, P. Musilek, X. Bai, (2001b), *Expressing Similarity in Software Engineering: A Neural Model*, Proc. SCASE-2001, Feb 8-9, 2001

Pedrycz, W., (2001c), *Fuzzy Equalization in the Construction of Fuzzy Sets*, Fuzzy Sets and Systems, 2001, to appear

Qiu, Guoping and A. W. Booth, (1996),. *Frequency sensitive Hebbian learning*. In ICNN 96.

The 1996 IEEE International Conference on Neural Networks (Cat. No. 96CH35907), volume 1, pages 143-8. IEEE, New York, NY, USA

Richard, Jones, (1999), *Introduction to MFC Programming with Visual C++*, Prentice Hall, 1999

Rojas, R., (1996), *Neural Networks, a systematic Introduction*, Springer, 1996

Suganthan, P. N., (1998), *Hierarchical Overlapped SOM Based Multiple Classifiers Combination*, The Fifth International Conference on Control, Automation, Robotics and Vision(ICARCV'98), Singapore, 9-11 December, 1998

Suganthan, P. N., (1999), *Hierarchical Overlapped SOM's for Pattern Classification*, IEEE Transactions on Neural Network, V 10, NO. I, Jan. 1999

Zadeh, Lotfi, (1965), *Fuzzy Sets*, Information and Control 8:338-353, 1965

Zadeh, Lotfi, (1996), *Fuzzy Logic = Computing with Words*, IEEE Transactions on Fuzzy Systems, 2, 103-111, 1996

Appendix

1 SOM Tool Package

(Downloadable from http://www.ee.ualberta.ca/~bai)

Executable file: SOM.exe

Sample data files:

1. synthetic.txt---a two dimensional synthetic data set. There are two distinct clusters in this data set.
2. MIS.txt---a medical imaging system data set(Munson, 1996).
3. jdk.txt---from JDK library, using OO measures.
4. Software.txt---from a java project, with 150 classes, using OO measures.

Example result files:

1. Demo150Nov21.som --- the training result for "Software.txt"
2. jdk20x20_Nov18.som --- the training result for "jdk.txt"
3. mis10x10_Nov18.som --- the training result for "MIS.txt"

Main Usage:

1. Extract all the files into a directory.
2. Run SOM.exe.
3. Open an existing "*.som" file or click "Browse..." to pick a data file, which should have the same format as the sample data files.
4. Set the training and displaying parameters, or simply use defaults.
5. Click "Train" to begin training, or click "Display" to display the maps.
6. During training, you can click the tool bar button to pause or stop training.
7. When training is stopped, left-click mouse on "Clustering Map" or "Data Distribution map" to pick interested cells.
8. Right-click mouse on "Clustering Map" or "Data Distribution Map" to select further analysis on the selected data.

Training Data Format

The training data must be plain text and is delimitated by space, comma or table. The first row and the first column are essential. The first row must be the column headers which are text descriptions of the corresponding columns. The first column must be the name of each data record. A sample Java software measure data set is shown below:

Class	LOC	NOM	DIT	NOC	CBO	RFC	LCOM			
linguist.basic.BLErrors	11	1	0	1	11	55				
linguist.basic.BLGetCondition	59	1	3	0	22	28	0			
linguist.basic.BLGetPair	8	2	3	0	5	5	1			
linguist.basic.BLGetValue	138	4	3	0	41	59	6			
linguist.basic.condition.BCContains	6	2	2	0	2	5	1			
linguist.basic.condition.BCEndOfFile	6	2	2	0	2	3	1			

2 *Screen Shots for SOM Tool*

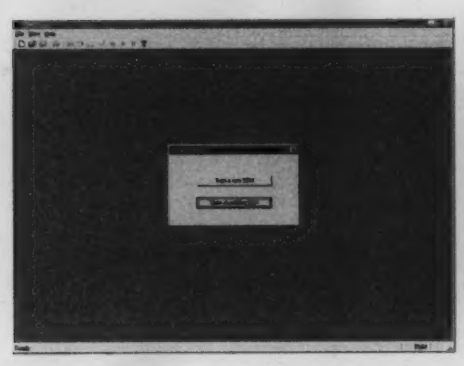


Figure A.1 Startup Screen

Figure A.2 Open Existing File Screen
The *.som files are the training result files

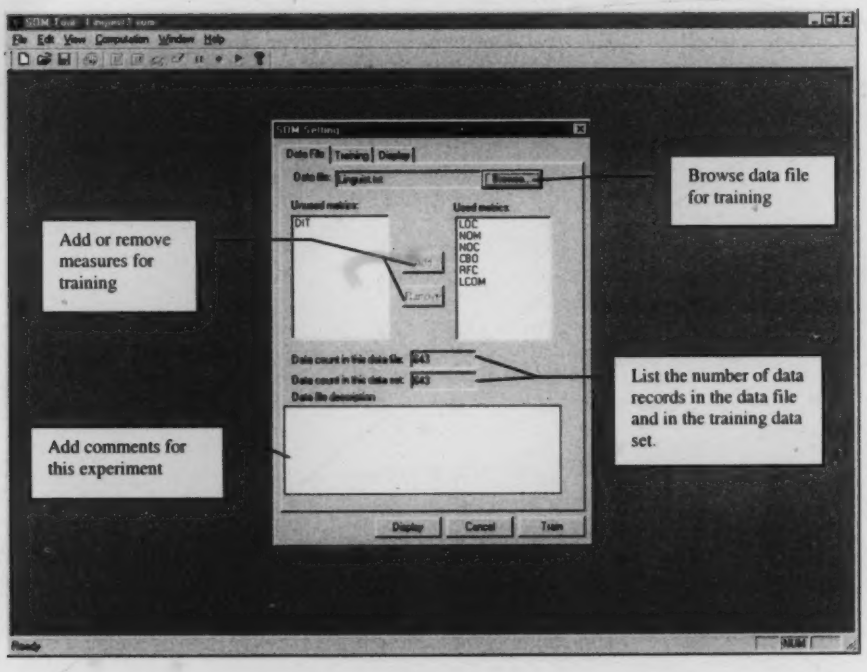


Figure A.3 Training Data Setting Screen

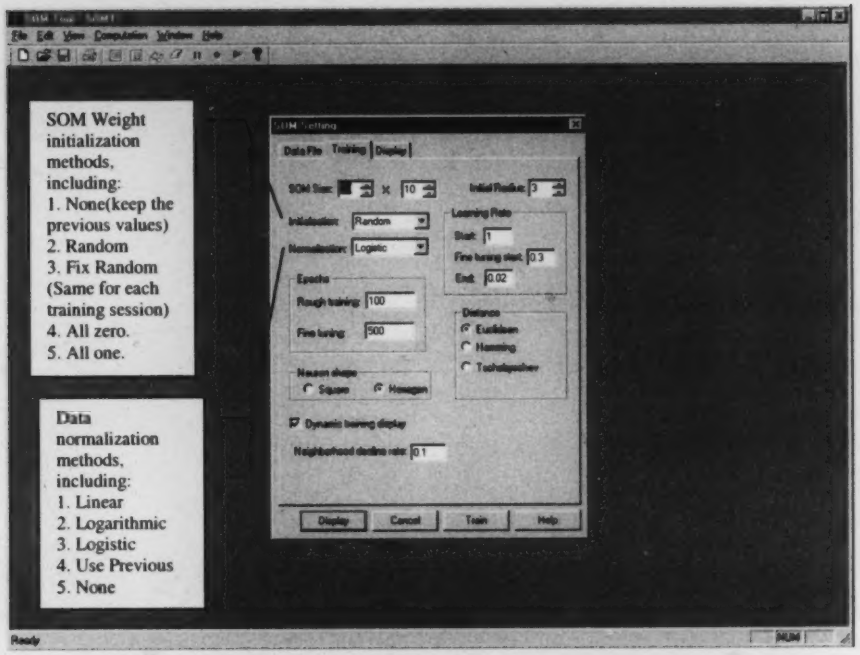


Figure A.4 SOM and Training Parameter Setting Screen

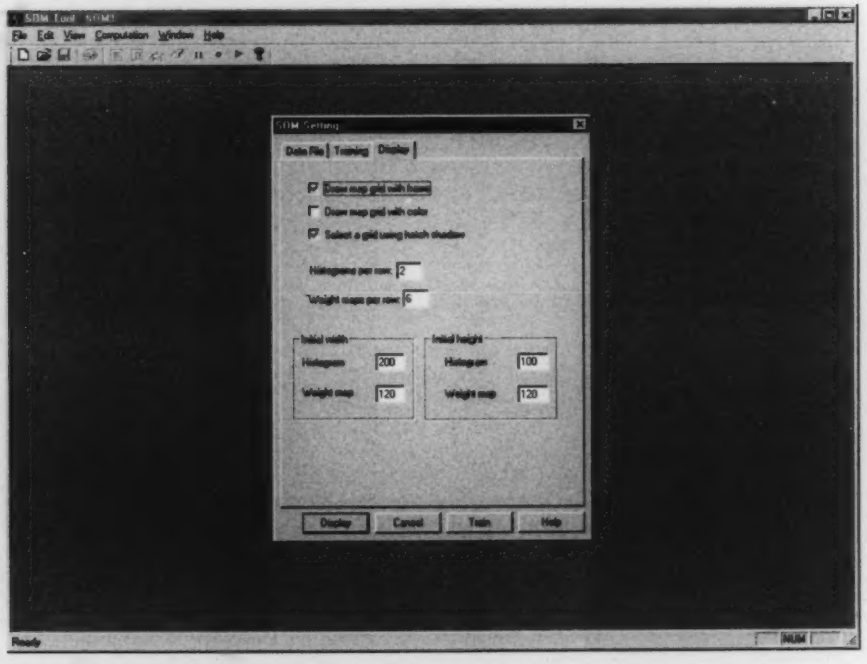


Figure A.5 Map Displaying Setting Screen

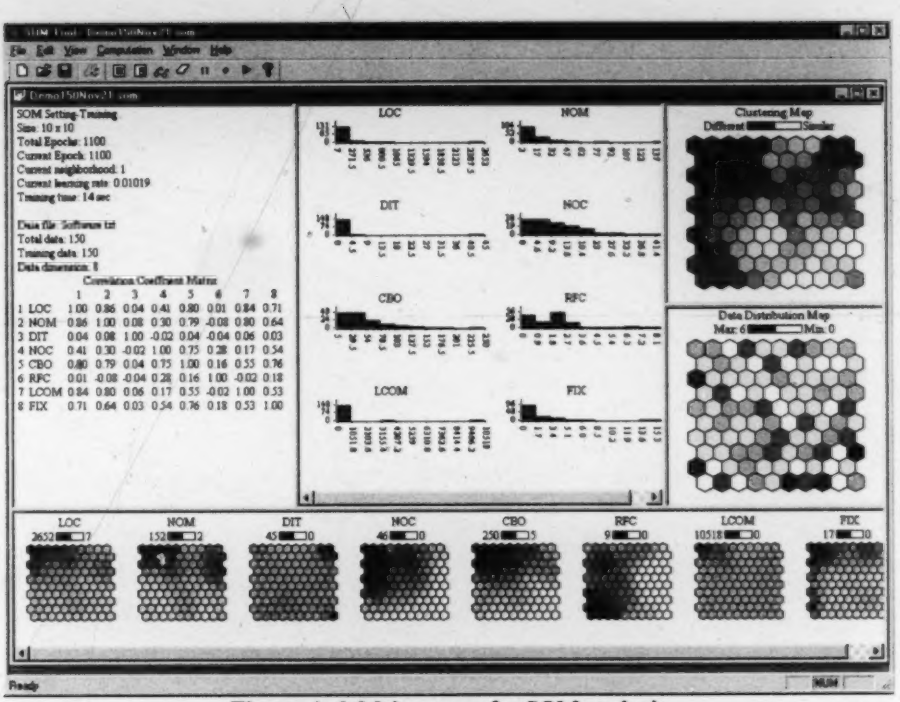

File Edit View Computation Window Help
SOM Setting-Training
Size: 10 x 10
Total Epochs: 1100
Current Epoch: 1100
Current neighborhood: 1
Current learning rate: 0.01019
Training time: 14 sec
Data file: Software.txt
Total data: 150
Training data: 150
Data dimension: 8
Correlation Coefficient Matrix
1 LOC 1.00 0.86 0.04 0.41 0.80 0.01 0.84 0.71
2 NOM 0.86 1.00 0.08 0.30 0.79 -0.08 0.80 0.64
3 DIT 0.04 0.08 1.00 -0.02 0.04 -0.04 0.06 0.03
4 NOC 0.41 0.30 -0.02 1.00 0.75 0.28 0.17 0.54
5 CBO 0.80 0.79 0.04 0.75 1.00 0.16 0.55 0.76
6 RFC 0.01 -0.08 -0.04 0.28 0.16 1.00 -0.02 0.18
7 LCOM 0.84 0.80 0.06 0.17 0.55 -0.02 1.00 0.53
8 FIX 0.71 0.64 0.03 0.54 0.76 0.18 0.53 1.00
LOC
NOM
DIT
NOC
CBO
RFC
LCOM
FIX
Clustering Map
Different Similar
Data Distribution Map
Max: 6 Min: 0
Ready

Figure A.6 Main screen for SOM analysis

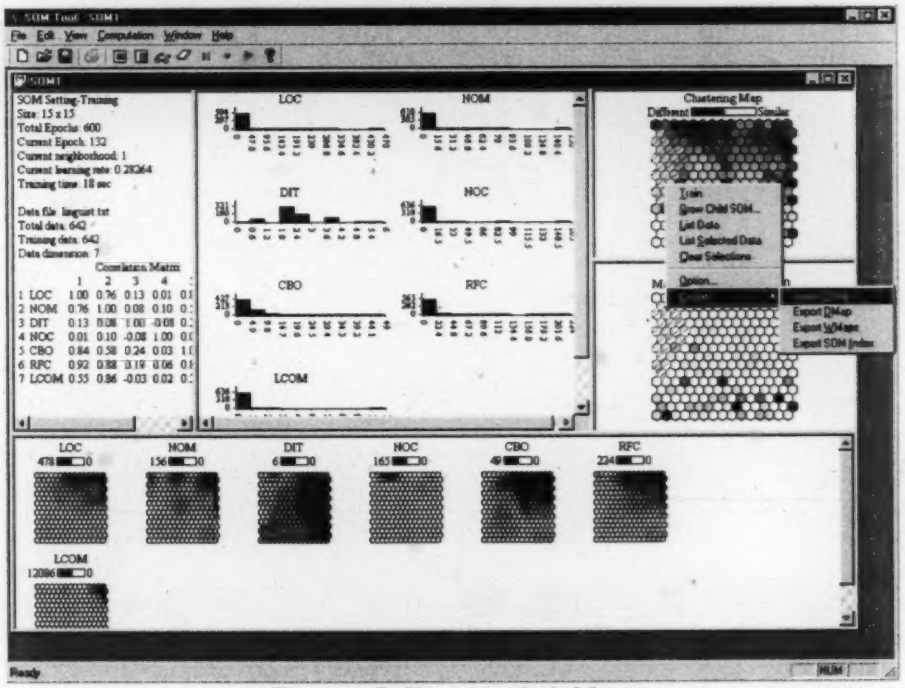


Figure A.7 Cluster Analysis Menu

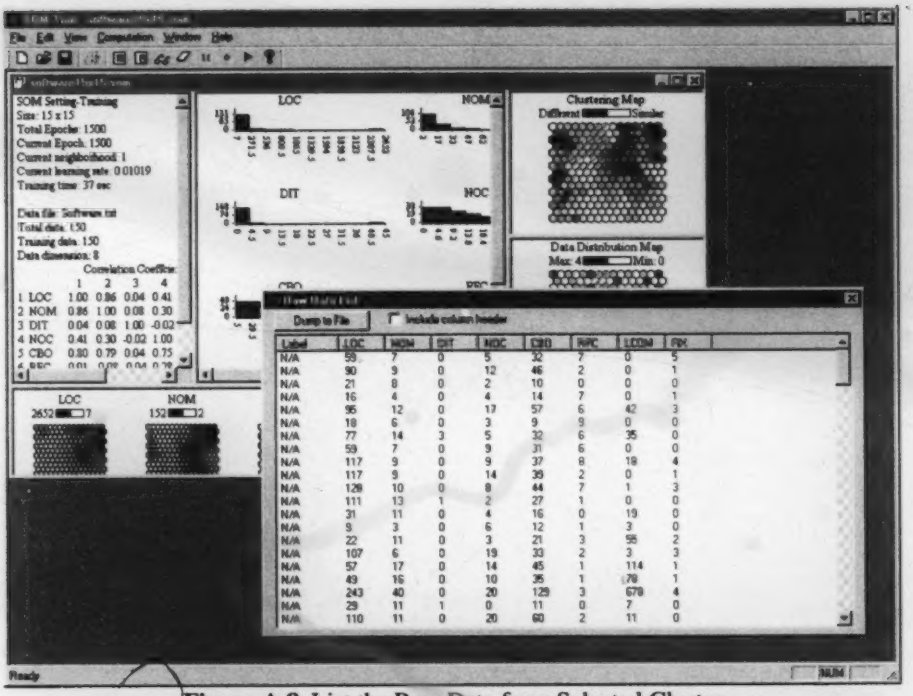


Figure A.8 List the Raw Data from Selected Cluster

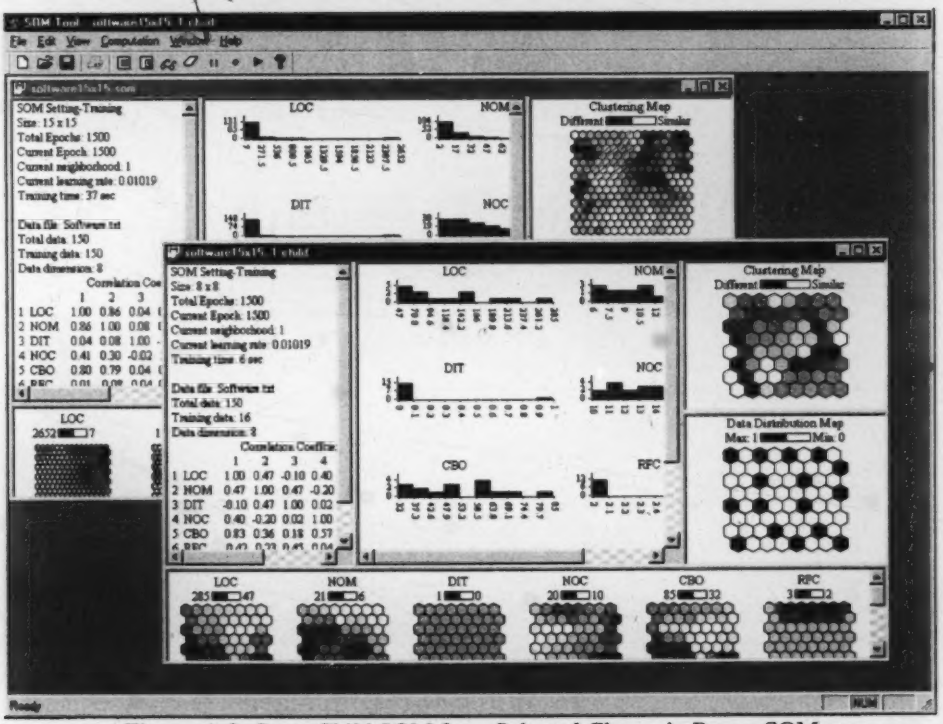


Figure A.9 Grow Child SOM from Selected Cluster in Parent SOM